Source Code & Software Patents:
A Guide for Attorneys & Experts to Software & Internet Patent Litigation

Andrew Schulman
http://www.SoftwareLitigationConsulting.com
undoc@sonic.net

Submitted for GGU IP LLM 386 (Fall 2013, Prof. William Gallagher)

Introduction

This is a law book about facts. It narrowly focuses on one particular type of evidence, so-called computer "source code," and how this evidence is employed in one particular type of legal case, litigation alleging infringement of software- and internet-based patents.

Source code is text – one or more documents – written by computer programmers. Every software-based service, device, or product used by consumers has been built from source code, often entirely from source code. Written in human-readable programming languages (these languages have names such as "C++", "Java", or "PHP"), source code is mechanically translated into the machine code required for the program's execution by a computer.[1]

Unlike a piano roll,[2] for instance, which tells a specific type of device (a piano player) what specific music to play, source code contains instructions to an essentially "universal" general-purpose device (a

[1] This mechical translation is itself carried out by another computer program: a "compiler," "interpreter," or some combination of the two. See ch. xxx.

[2] [maybe cite software © cases analogizing to piano rolls?]

computer), telling it exactly how to take on the attributes of a more-specific device. Software is largely[3] the reason why a single piece of hardware such as an iPhone can act as an MP3 music player or Netflix viewer one moment, a word processor the next,[4] or as a camera, game-playing device, email reader, text messenger, or web browser.

The ability of a single general-purpose physical device (computer hardware in some form, be it a laptop or a mobile phone) to mimic a host of specific devices springs from the combination of an attribute of the hardware with an attribute of the software. The hardware contains a microprocessor (made by a company such as Intel or Qualcomm), whose job is to rapidly read and then carry out ("run" or execute) instructions. The software contains these instructions. While the microprocessor cannot read source code, it does read a somewhat different form of software (so-called machine language) which is mechanically compiled or interpreted from, and which bears a direct relationship to, the source code written by humans. Source code, and software generally, are both forms of text; source code is more readable by skilled humans.

Software, in other words, is text describing the operation of a machine. Whereas a piano roll or an MP3 file or a word processing document generally provide input or parameters to that operation, and are therefore considered *data*, software describes the operation itself, and is therefore considered *code*.[5] This is not "code" in the sense of secret code or cipher broken by the NSA, but code in the sense of something that stands in for something else: in this case, a numeric code which a machine reads and then uses as an indication of what specific operation to perform.

Source code & Patent claims

As documents containing instructions which describe or comprise a particular device or set of operations, source code bears more than a passing resemblance to patent claims, which are single sentences using sets of elements or steps to describe the boundaries of an invention. It is true that, whereas a properly-written patent claim defines only the boundaries of a device, system, or method, software must spell out (implement) everything, not just the boundary that distinguishes this code from all other code, in excruciating detail. However, any given piece of source code will likely only spell out its

[3] Of course, the hardware must contain the requisite real-world components, such as a screen (possibly touch-sensitive), audio capabilities, and light-sensitivity capabilities (used to software to mimic a camera).

[4] [actually, at nearly the same time: multitasking]

[5] Yet a key aspect of modern computers is that code can also be viewed as data. This has crucial implications for the ability to reverse engineer software; see ch. xx. As a less important caveat, note that some types of data files can also contain code, e.g. macros, formulas, and fields inside Word documents and Excel spreadsheets. [Perhaps this is place to note that the e-discovery (ESI) field focuses almost entirely on data rather than code, and that e-discovery people, like IT departments, seem to not know much about code. Partly because data will often be more relevant, but still...]

instructions to a machine at a certain level of detail, leaving lower-level details to other pieces of source code, or to the machine itself.[6]

Computer science has proven the equivalence of machines and languages.[7] While largely a practical, hands-on guide, this book will touch on the implications of this remarkable equivalence, and will suggest implications for the patent system of the facts that machines can be precisely described in language, and that today many machines in the real world are brought into being through that descriptive language. Whatever the fate of software- and internet-based patents as such, or business method patents, a 21st century patent system is inconceivable without some intimate connection to software.

Of course, there is one enormous difference between source code and patent claims: whereas a granted patent claim merely describes (albeit in a legally-operative way) a device, system, or method, source code in contrast bears a direct relationship to *implementing* the device, system, or method. The language of source code can be viewed as an especially precise form of claim language, which results directly in devices or methods used in the real world. A core theme of this book is the relationship between the language of source code on the one hand, and the language of patent claims on the other. How to bridge the gap between these two types of language is the crucial skill required by an expert in software patent litigation, and should inform much of what attorneys must do in these cases.

Source code is the primary form of technical evidence used in software patent litigation. During discovery, each side will request access to the other's source code, and experts or consulting examiners will then be tasked with inspecting (under a protective order) the source code. Frequently, the source code is kept on a tightly-controlled computer only accessible at normal business hours at a single location (conveniently located thousands of miles from your expert). Parties will generally need to examine their *own* source code with an "outsider's approach" similar to that taken with the opponent's source code. Source code relevant, for example, to anticipatory prior art may also be subpoenaed from third parties.

Because it is text that can directly yield a good or service, one may think of source code as an odd type of blueprint or memo that can be made immediately operative, somewhat in the way that the words "I agree" can bring into being a contract.[8] Another dominant theme of this book therefore will be the somewhat odd relationship from source code to the revenue-generating products and services that are presumably the primary concerns ("show me the money") of patent litigants.

---

[6] [Ideally, a given piece of code would only contain what uniquely distinguishes it all from other code, and each such unique piece of code would be callable by other code; cf. "impossible dream of software re-usabiity."]

[7] [cite Turing et al., note much of this work was done before (and as part of initial design for) the existence of physical digital computers: computer science before computers (Curry, Church, etc.)]

[8] In addition to legally-operative words, a loose analogy might also be to speech acts (Searle, Austin).

Wider applicability of source-code examination

Apart from the intrinsic (and perhaps growing) importance of software and internet patents, and therefore of the core evidence used to prove or disprove their infringement or validity, this book's in-depth look at source code, and at how experts compare it with the different form of text comprising patent claims, is also intended to illustrate larger points about source code as evidence.

This one type of evidence is important because nearly all actions attributed to computers, to the internet, to mobile devices, or to any device such as a camera or a car containing one or more chips – in other words, anything involving a large portion of what for better or worse makes up daily life in the so-called and somewhat absurd developed world – are actions directed by, or even carried out entirely within, software, which was first written by a skilled human in the form of source code.

Apart from the obvious role of software in other forms of intellectual property (copyright,[9] trade secrets, and even to a small extent trademarks)[10] and antitrust (e.g. the focus of US v. Microsoft on whether Microsoft's web browser was "integrated" with its Windows operating system) -- the design and implementation of software has also be relevant in:

- white-collar crime (e.g. the "House 17" program used by Bernie Madoff to generate fake trading data),[11]
- products liability,[12]
- corporate or employment law (determining a corporation's de facto policies may hinge on what its in-house software actually does, as opposed to what a manual says),[13]

[9] [what the software actually does, while less central to © than to patent law, is nonetheless at issue when there is non-verbatim copying; two types of source code (perhaps written in different programming languages, for different platforms, etc.) must be compared with each other under A/F/C (which given the volume of code in most cases, will likely really be C/A/F).]

[10] [e.g. trademark employed as technological lock-out device, e.g. "SEGA" boot ROM in TMSS (Sega v. Accolade); "Nintendo" TM in code; "IntelInside" in CPU registers; possibly code required to use a technical-standard certification mark (e.g., "USB")?]

[11] [cite US v. Jerome O'Hara and George Perez, two programmers who worked for Madoff]

[12] [cite Cem Kaner, Bad Software, and something more recent]

[13] [cite some workplace privacy case with company's de facto software policy vs. its de jure policy known to HR manager; perhaps just illustrates the general point to not expect higher-ups in the company to actual know what it does?]

- administrative or environmental law (e.g., cases demanding to see the EPA's computer models and simulations supporting an alleged over- or under-regulation).[14]

At a level below that of cases where source code can be directly relevant, there is what may be called "software forensics."[15] A currently-prominent example is the demand made by some defendants in drunk driving (DUI) cases for inspection of the source code for the Breathalyzer device which was used to measure their blood-alcohol level.[16]

Now, inspecting the source code for a Breathalyzer in a DUI case seems like a stretch into just-barely-reasonable doubt (clearly it would be central in a case involving a patent for digital breath-alcohol monitoring devices). Output of a system is usually authenticated under FRE 901(b)(9), not with an in-depth walk-through of the system's design or implementation, but rather through simple testimony that the system is in fact *relied upon* in some non-litigation context.[17]

But there is an underlying point which is important. Everything referred to as "computer-generated evidence"[18] is in fact the output of *software*. Software is written by humans, or at least by computer programmers. The output produced by all the digital devices we so rely on and take for granted is not quite like the output of non-digital devices like old-school copy machines, cameras, or typewriters, whose output was once generally admitted with nary a thought given to the device that produced the output.

True, those devices too were designed by humans, and subject to error or even manipulation. But every photograph is not hearsay spoken by the camera designer.[19] The role of the camera's designer is far too attenuated for that. In any case, courts don't want, and the justice system could not handle the burden of, in-depth examination of every device whose output has been introduced in court. Instead, the

---

[14] [cite Sierra Club v. Costle; etc.; maybe even note constitutional law cases hinging on "constitutional facts" (Fiagman) of what constitutes discrimination, which in turn depends on sophisticated statistical models, which are basically software; a bit of a stretch, but see "Math on Trial", Tribe, etc.??]

[15] [distinguish software forensics from computer forensics, and both from what source-examiners do in patent litigation; forensics almost always involves individuation (connect this evidence to this specific person), whereas "software forensics" properly defined should probably be limited to uses like the DUI cases, or tracing software to its author (cf. Bob Slate book?). Also contrast locating perp for internet attack (cf Internet Forensics book) which generally looking at data, not code. Try to limit "software forensics" to examine the code, not data, in a forensics context, which may include establishing reliability of devices used in criminal investigation.]

[16] [see e.g. Jennifer Mnookin "Of Black Boxes, Instruments, and Experts: Testing the Validity of Forensic Science" (http://www.law.yale.edu/documents/pdf/Alumni_Affairs/Mnookin_Of_Black_Boxes_Episteme.pdf) pp. 10-17: BREATH TESTS FOR THE DETECTION OF ALCOHOL AND THE PROBLEM OF SOURCE CODE]

[17] [cite 901(b)(9) cases rejecting need to examine system's source code]

[18] [distinguish computer-*stored* evidence]

[19] [but see early photo & video cases (cite "Mechanical Witness"), and note two theories of photo admissibility: besides "fair depiction," there is also "silent witness" theory which can get into details of operation]

system requires quick *proxies* for reliability,[20] such as "fair depiction" testimony for photographs, or some quick testimony to a business's routine day-to-day reliance upon the system which has generated some printout now being introduced as evidence.[21]

So only the most zealous criminal defendant's advocate would question the authenticity of most of the output generated by software. Or, because it is in some remote sense the output (statement) of a computer programmer, try to have it ruled as hearsay.[22] The doctrine denying the existence of "machine hearsay" rightly makes it difficult to bring such challenges.[23]

And yet, there *is* something wrong with applying the denial of "machine hearsay" to digital or electronic output brought about in large part by software. The human role in digital devices and their output is more significant than in most older analog devices. While everything generated by computer should not therefore be dubbed "hearsay,"[24] and excluded as evidence until some exception is found for it (with the system-reliability exception presumably modified to demand testimony by the programmer, or an expert skilled in programming), nonetheless much of the evidence which now finds its way into court ultimately rests on source code, and at least requires a better basis for authentication and admissibility (or *weight*) than the doctrine which states that machines do not generate hearsay, or that anything generated by a computer is sufficiently reliable (or at least, authentic) by virtue of the computer's use apart from this litigation.

At the very least, this large category of evidence should be thought of as "software-generated evidence" to force courts and attorneys to stop and consider that there is not only a machine but also a human and language behind it. Daubert and its progeny teach that some evidence should not be accepted at face value, or on the mere fact that it has been relied upon somewhere else.[25] Similarly, following the National Academy of Science's 2009 report on forensics problems[26] (highlighting the lack of standardization, accreditation, and certification even in established fields such as fingerprint matching and hair/fiber forensics), fields of expertise employed in both criminal and civil contexts have been

---

[20] Much as trademarks play the role of proxies, to avoid close inspection of goods which would otherwise be required by "caveat emptor." [cite]

[21] [this is far more problematic than made to sound here; what about systems whose only purpose is generating forensic output?; reliability should not then be determined based merely on having been relied upon in some non-litigation context, because there is no such context. Cite xxx]

[22] [thereby requiring an exception such as FRE 803(6) (business records), with the regularity of the underlying activity called into question]

[23] [cite "machine hearsay" case, and one noting that "output" of a sniffer dog is not hearsay]

[24] [but see Rice, Foundations of Digital Evidence]

[25] [note similarity of Frye "general acceptance" (which plays larger role in Daubert than might be thought) with finding reliance, not by delving into how something works, but from proxy that it's already been relied upon in some other context; also note the polygraph-like device in Frye as the canonical machine producing litigation-relevant output]

[26] [the title "Strengthening" was a polite way of putting it?]

challenged to clarify the methods that experts use in selecting and generating facts, and in moving from those facts to conclusions. Under Kumho Tire, indicia of reliability, such as a known error rate and repeatability, apply to engineering/technical as well as scientific evidence. The Daubert challenge applies not only to expert testimony, but also to tangible evidence.[27]

This book therefore has a slightly-hidden agenda of "defamiliarizing" the output the computers, through wider examination of source code.[28] It is of course the role of experts, especially those retained by defendants, to defamiliarize, to make the commonplace problematic, and to question the reliability of facts introduced by the other side, who in turn must be prepared for this approach of turning seeming mountains into molehills (see ch. xx).

The point here is not the well-worn "garbage in, garbage out" (GIGO) phrase applied to those darn computers.[29] Nor is it to suggest that software in general, or source code in particular, is somehow *sui generis*. Though software does possess some novel and even remarkable attributes (text which directly tells a machine how to mimic the behavior of another machine is not just a "piano roll"), it is just as important to *demystify* software as to call into question or defamiliarize the output generated by software.

Indeed, another goal of this book is to reduce the mystique or aura that seems to surround source code. This mystique has resulted in what are frankly bizarre protective orders (POs) being accepted as the norm.[30] As noted earlier, this may in part be due to a misunderstanding of the word "code" in "source code." Again, code in this context does not mean secret; it means a number that a machine can interpret as an instruction. Of course, a given piece of code may be valuable and secret, and even constitute the company's "crown jewels," but it is hardly so by mere virtue of being in source-code form.[31]

Source code is best thought of as just another "doc," with a difference. It is a document capable of producing direct results, and is a crucial foundation for Our Modern World, but it is still a doc, which can be read by those skilled in the art.

---

[27] [cite]

[28] [Given that I am a source-code examiner, this just sounds like special pleading: growing importance of software blah blah, more evidence (whose basis ultimately is source code) should be called into question, etc.; right, like anyone ever writes a book thinking the subject is becoming less important.]

[29] [cite Johnny's Oyster & Shrimp case referring to entire internet as unreliable source of gossip]

[30] [cite AJL article on overly-protective source-code POs]

[31] [This "aura" around src code also yields incorrect results in TS cases, where blanket/umbrella designations of all src as "crown jewel" results in failure to demarcate where specifically a valuable secret, which gains its value from fact of secrecy, might reside among say one million source code files, many of which were adopted verbatim from open source. But: compilation as whole could be TS or confidential, even when component parts are even entirely non-secret; cf. database compilation cases]

Specifically when do these odd documents need to be read? That they are the central evidence in software and internet patent infringement cases can be taken as a given. That they need to *NOT* be read, nor the systems built from them called into question, in any but exceptional non-IP cases is also clear. But what lies between these extremes – the cases where the operation of a software-based digital system should be called into question and its source code examined – is an useful question, an answer to which will emerge in the course of this book.

One way in which this can be indirectly answered is by asking, in the context of software patent litigation, how far down one needs to drill into the accused instrumentality or the assertedly-anticipatory prior art. Even when a large body of source code is relevant to a given patent claim, *proportionality* is an absolute requirement that is often ignored in the name of "zealous advocacy" (or worse).[32] In ch. xxx on discovery, and indeed throughout the book, the question "How deep down can we afford to drill?" will be raised. The focus will largely be how *much* source code one can afford to discover and examine, but it implicates [yuch] the larger issue of how much the legal system can afford to delve into technical questions, balanced against what is likely a greater need than currently realized.

Patent Litigation as a Model for Source-Code Examination

In other words, whenever a balanced but skeptical attitude towards software-generated evidence calls for source-code examination, then one model for this type of examination is software patent litigation. Not "model" in the sense that the current practice should be adopted (for it almost surely requires some reforms), but meaning that an in-depth look at the current role of source code in software patent litigation should be summarizable in a set of points which can be accepted or rejected in non-patent contexts. Proper methods for examining source code in patent litigation should lend themselves to assessing software-generated evidence.

Why look to one area of the law, rather than to the software industry or computer science, for these methods? Certainly source-code examination and software reverse-engineering are commonly practiced in the industry.[33] However, for a variety of reasons little is written down about it. The software field thinks of itself as producing software, not sitting around looking at it. This despite the common knowledge that the vast majority of software development involves maintaining existing code rather

[32] [And note that the other side may be happy to give you everything you've asked for, and more; cf. "dumping" -- "careful what you wish for; you just might get it"]

[33] [cite]

than writing new code from scratch.[34] So yet another goal of this book is to codify tacit knowledge regarding source-code inspection, to suggest.[35]

Inspection of source code in patent litigation should provide some guidance in thinking through broader questions of the role of technology in law, which ultimately involves the question of how society as a whole can get its arms around what we have come to depend on. [yuch vague, but try to get this right; other examples where modern IP is seen as informing non-IP parts of law?; patent law as THE modern form of boundary drawing?; patent system as a way not only of protecting and over-protecting tech but also as a way of classifying and organizing it (even if not doing a very good job of disclosing it).]

Certainly, if Lawrence Lessig was correct in asserting that, at least in so-called "cyberspace," "code is law,"[36] and if others are correctly noting a trend to replacement of legal with technological restrictions,[37] then some methods of determining this law or quasi-law (i.e., de facto policy, based on the defaults created by software) is needed, apart from Lessig's advocacy of "open source" as the apparent solution to the problem of decisions, policies, and standards being embedded in software. Even "open source" must be closely read, and proprietary non-open code (the type whose source code is requested and produced in discovery in patent cases) is likely to be with us for long enough to require standard inspection methods.

Specifically, what is now several decades experience in patent litigation with matching source code to patent claims -- determining if one text "reads on" another -- provides an important foundation for what will likely be an increasingly important need to assess software. Assessment or inspection is usually a matter of comparing one thing against some "exemplar."

Unfortunately, the experience with this matching is largely uncodified, even in published opinions, which naturally focus far more on what the claims mean (Markman) than on whether they do or do not match the accused instrumentality, this being of course within the jury's province. Even opinions regarding each party's "claims tables" (which align source code in one column with claims elements or steps in the other column) quite properly avoid the technical details of how one decides whether code containing X/Y/Z matches a claim containing A/B/C.

But matching claims with code is not so immediately intuitive or straight-forward that is doesn't rest on *some* methodology.[38] If bar-exam graders can be calibrated to arrive at very similar scores for a written

---

[34] [This is too strong; lots of material on examining code for security problems; for walk-throughs; debugging; etc. But odd that e.g. only one book on "Code Reading" as such (Spinellis).]

[35] [Intended to suggest a protocol or standard for source-code examiners to use, as a step towards agreement on "best practices" for source-code examination. See IEEE Standard for Software Reviews, IEEE Std. 1028-1997: anything specific; other IEEE or ACM standards?]

[36] [cite; note Posner v. Lessig re: "horse law"; Lessig citing Mitchell, City of Bits]

[37] [e.g., replacing or augmenting contractual restrictions with technological barriers]

[38] [CodeSuite for © and TM; anything in digital forensics e.g. EnCase, CISSP, etc.?]

essay, or a large number of radiologists arrive at the same reading of an MRI,[39] surely standards can exist for code examination in patent litigation on issues such as how to determine whether X/Y/Z matches A/B/C, and what the significance of such a match might be. This book argues that there already is a standard for this type of matching, within the field of software patent litigation. However, the standard is tacit and uncodified, as in the software industry itself as noted above.

Whatever happens to software patents (and again, it is hard to conceive of a modern patent system without software patents, especially given the difficulty of distinguishing software vs. non-software patents, at least at the margin),[40] the process of matching code to other text will remain important in patent law, and law generally.

Themes

Most of this book is on a much more hands-on, non-"theoretical" level than what has just been said in this introduction. The book's recurrent themes are more at the following level:

- Because even literal infringement of a software patent is not a matter of verbatim copying, source code may indicate literal infringement of a patent claim even when the terminology used in the source code is entirely different from that in the claim. Thus, it is crucial to understand synonyms, and to understand that the naming used in code is both crucial and arbitrary (a term or name in software is a symbol standing for a numeric address). [rewrite this; it isabout as clear as mud]

- When comparing code with claim elements or steps, the comparison employs specific features or attributes of the code on the one hand and each element/step on the other. The comparison should not be a holistic exercise. Even for literal infringement, the comparison should step through attributes such as function (role), way (implementation details), and result (output).

- There is room for expert disagreement because code can be viewed from different perspectives, at different levels of explanation, and with different aspects of a generally accepted standard. For example, code should be examined both statically (read the code) and dynamically (run the code, with instrumentation).

- While code is distinct from data, code is also a special form of data, and as such can be manipulated in ways that make reverse engineering feasible. Indeed, software-based products can be surprisingly readable, even without source code; and inspection of source code can often be automated in various ways by using programs that manipulate source-code files.

---

[39] [see literature on inter-rater assessment]

[40] [cite multiple articles disputing with each other the definition of a "software patent"; note biotech, business models, finance, communications]

- Software products available on the market do not only contain the "1s and 0s" which make up machine code. They also contain large amounts of readable text ("strings"), which can be used to examine the product. This has important implications for pre-discovery Rule 11 obligations.

- Source code is generally NOT the accused instrumentality. It is the product which may or may not infringe. Source code is usually "mere evidence."
- It can be difficult to tell the difference between the experts' disputes over what the source code does on the one hand, and disputes over what the claim means on the other. [yuch; fix]

- Small non-technical words in patent claims, or suffixes such as –ed and –able, can make all the difference.

- Source code can be expected to answer or help answer questions regarding infringement, non-infringement, anticipation, and statutory bars. Other questions include remedies/damages, obviousness, sufficiency of disclosure, other reasons for invalidity (including inventor naming, laches, and standards committee participation), and source code may be an important evidence here too, as covered in ch. xx on Goals of Source Code Examination. A significant part of the expert's task is taking these broad legal questions and disaggregating them into a larger series of smaller technical questions which source-code inspection can answer. [For example...]

The dozens of patent cases discussed in this book revolve around a relatively small of issues, such as:

- Pinpointing how: Claims tables asserting that source code shows infringement of a patent claim must, once source code is available in discovery, "pinpoint" where in the accused instrumentality the patented invention is found. Aligning each claim element or step with a "see this function in this source-code file," even with specific line numbers, is insufficient, because "see" doesn't really say anything. Some courts require "how" as well as "where": explain how each of A/B/C in the code is the same as X/Y/Z in the claim.

- Latent code: Not everything in the source code for a product is actually part of the product itself. Not everything in the product is ever executed. Whether such "latent code" infringes depends on the type of claim (apparatus v. method) and the type of infringement asserted (use vs. make/sell/etc.).

- The laundry list: a claim is not a "bag of parts" or "laundry list" (except perhaps in the rare case of a patent for an unassembled kit). Thus, it is not sufficient to find X, Y, and Z in the source code. They must fit together into an integrated whole; connectives between elements or steps are often crucial. These connectives are often vague-sounding. [give example or cite ch. xx; maybe cite ch. xx for each of these points?]

- "Inherent" infringement: Associating A/B/C in the code with X/Y/Z in the claim shouldn't depend on an expert's assertion that A "inherently does" X. It must be shown that A is X or its

equivalent. [sounds vaguer than it is; give example, or refer to ch. xx]

- Representatives: When there is a large number x of accused products, it may be possible to produce fewer than x claims tables. However, any products chosen as "representative" of a larger family must be shown to be representative.

- Burden of explanation: In some circumstances, it surprisingly may be defendant's burden to explain how its code works, even (though rarely) having to align its code with plaintiff's claim elements/steps.
- Prior art: source code may constitute anticipatory prior art, but it often will not when the source code is proprietary, and "only" the product was made public. The source code may however be evidence for what the product may have publicly disclosed.

- Protective orders: source code as a whole will generally be given an umbrella/blanket protective order, without consideration for demarcation of specific confidential or trade-secret information within the source code.

- Discovery requests: be reasonable; don't ask for all source code for all products going back ten years; don't refuse under any circumstances to produce source code.

- "Missing" source code occurs frequently, but is not always spoliation. Even a company asserting that all its source code constitutes the company's "crown jewels" may truly be missing some for a non-malicious reason. On the other hand, the cases include some remarkably egregious examples of source-code destruction and alteration.

- Third-party source code: Limits on the quantity of source code to be produced, or even on whether source code is necessary at all to the case, are far more likely to be imposed when requested from a third party.

Audience

This book is intended for attorneys, experts, and non-testifying consultants performing source-code examination:

- For the software expert or examiner, the book provides questions to ask yourself, similar to what the other side might ask in a rough deposition, to make sure you didn't forget something, and to provide the confidence that comes from using a neutral, considered methodology. Hopefully technical readers will nonetheless at least skim the legal sections, to see where their work fits into the overall goals of the case.

- For attorneys, the book should help you select and work with your own expert, to know what you should and shouldn't expect from source-code examination, and to engage in discovery, claim construction, and other legal tasks, in a way which meshes properly with the source-code exam. Many readers will be patent attorneys, and therefore ex-engineers, presumably with some college contact with programming. However, programming knowledge is not assumed, nor will this book teach attorneys how to read source code, except perhaps in a superficial sense useful in a deposition. Rather, the attorney will learn how to work with their expert. [No, more than that; also source-code discovery, PO, etc.] Hopefully attorneys will at least skim the technical sections to see what makes up a source-code exam.

- Opposing attorneys should find the material useful in deposition and cross-examination of the other side's expert: did you do X? if not, could that have changed your opinion?

- For multiple experts/examiners working on a team, ch. xx can help check each's other results, as part of quality assurance, or (presumably undiscoverable) devil's advocate, second opinion, or "team B" approaches.

Both plaintiffs and defendants

This book is intended for those hired by defendants as well as by plaintiffs. While D's burden is not simply the opposite of P's (D is not required to prove non-infringement, but rather to rebut P's sufficiently particularized assertions of infringement), D will usually need to inspect its own code in a way that closely parallels how P will inspect it. For reasons detailed in ch. xx, D will also often need to examine P's source code (even in the case of so-called "trolls" or non-practicing entities),[41] and that of third parties with potentially anticipatory prior art. Throughout the book, questions and tasks have been phrased to the extent possible in D/P-neutral form.[42]

Accessing your own client's code will typically be simpler than accessing the other side's, which is likely covered by a protective order significantly restricting time and place of access. At the same time, the producing party's expert should generally take an "outsider's" view of this code,[43] for example using the same tools that the opponent's expert will use, rather than relying solely on the in-house development

[41] [NPE will have bought patent from somewhere, and along with it often has source code, or access to the inventors who generally will still have source code (but: i4i)]

[42] [E.g., "that which infringes if after; anticipates if before" (if also publicly accessible & enabling) – thus, D can turn P's accusation against it, if P reads its claims as broad enough to read on D, then it is more likely broad enough to be read on [?] (anticipated) by prior art.]

[43] [explain why can be more desirable than insider's "straight from the horse's [client's] mouth" perspective]

tools. This also applies to third-party source code, to the extent subpoenable,[44] and to open source (as discussed in ch. xx, often the opponent's source code will contain significant portions of open source).

Roadmap

The bulk of the book marches through the steps of the source-code examination itself: preparation before the exam (Part II), how to conduct the examination itself (Part III), and how to afterwards use the fruits of the exam (Part IV). Most source-code exams require multiple visits to the "source code machine," and naturally some steps from Part III will be performed before some in Part II.

Part I of this book takes up some of the topics addressed above – what's so special and not-so-special about source code (ch. 1), and the relationship between source code on the one hand and commercial software products on the other, including the "latent code" cases (ch. 2). Ch. 3 then walks through the types of questions that can and cannot be answered with software, using both the journalistic who/what/when/where framework and the elements to be proven for patent infringement assertions and defenses, and issues such as damages, non-infringing alternatives, importation (ITC), and practicing/working. Ch. 3 also covers the contents typically found in a source-code production.

Part II, "Preparation for the Source-Code Examination," begins with examining software products without source code, particularly as part of the reverse engineering required for a reasonable pre-filing investigation under Rule 11 (ch. 4); this also includes preliminary infringement contentions (PICs). The next three chapters cover source-code discovery largely under FRCP 34 and under Local Patent Rules (ch. 5); the problem of "missing" source code and spoliation (ch. 6); and protective orders (POs), including what has been called the problem of "over-protective orders" (ch. 7). The two final chapters in Part II cover experts: ch. 8 on the expert's own preparation for the source-code examination (including a potted guide to claim construction for non-attorneys), and ch. 9 on the role generally of experts, including non-testifying source-code examinations, in software patent litigation.

Part III focuses on the source-code examination itself. After ch. 9 on the purpose of and general approaches to the source-code exam, and ch. 10 on the need as shown by the cases for diligent inspection (don't wait too long) and for careful adherence to the PO, the remaining chapters are largely technical, aimed at source-code examiners. However, the chapters on the nitty-gritty details of examining source code, and comparing it with patent claims, should be useful not only to source-code examiners, but also to those attorneys preparing to depose or cross-examine source-code experts ("did you do this? why not?"). It should also remind attorneys of why in the first place source code is being sought in discovery.

Ch. 11 proposes an initial orientation to the source code (birds-eye view), including tools available to aid the examination; ch. 12 covers testing for completeness, looking for missing code, and matching the source-code production to the accused product; ch. 13 is a very large chapter covering indexing and

[44] [cases]

searching source code, with special attention to searching for synonyms to terminology appearing in patent claims; ch. 14 discusses a tracing, which is different way to locate relevant code apart from searching; ch. 15 is on "close reading" of the potentially relevant source code which has been located through searching or tracing; ch. 16 is another very large chapter, on comparing code with claims, i.e., with the central task of reading claims onto the code; and touching on the general problem of forensic mapping/matching/comparison.

Having now at least provisionally aligned the source code with the patent claims, ch. 16 continues with comparing this source code with the accused instrumentality, in order to look for "dead code" and "latent code" problems; ch. 17 proposes questions for the examiner to ask themselves in order to analyze their results; ch. 18 covers the problems posed by huge volumes of source code (especially in what is called a "quantity case"); ch. 19 is on the seemingly humdrum but important tasks of note-taking and printing under the PO; ch. 20 on testing/confirming the provisional source-code results while source-code discovery is still ongoing. Part III ends with a chapter on double-checking the results of the source-code exam (ch. 21).

Part IV, "After the Source-Code Examination," starts with use of the fruits of the source-code exam, which are likely print-outs of files extracted from a much larger production (ch. 22); and issues of authentication and admissibility (ch. 23). Ch. 24 is a lengthy chapter on producing claims tables, which align extracted parts of the source code with the patent claims; ch. 25 covers the use of source code in expert reports and affidavits; and ch. 26 on the use of source code in depositions.

Part V contains a concluding chapter (ch. 27) applying the lessons of source-code examination in patent litigation to larger areas of the law, as suggested briefly in this introduction.