Informal consulting interactions between apprentices and experts represent a little-studied but common collaborative work practice in many domains. In the computer industry, programmers become apprentices as they retool themselves to new computer languages, programming environments, software frameworks and systems. Our empirical study of consulting interactions has provided insights into the nature of this informal collaborative work practice.

We describe the variety of "hard-to-find" information provided by the expert, the incidental learning observed, and the pair’s strategies for managing joint and individual productivity. Given these observations, we discuss how computer-based tools could help apprentices encapsulate task context, switch among subtasks, facilitate collaborative interaction, and supplement consultants.
1 Introduction

In the computer industry, professional programmers must often retool themselves. A project change may require working with a new software system, in a new programming language, and using a new toolset. Modern object-oriented frameworks of reusable software components have increased the amount to be learned, since frameworks commonly have hundreds of interdependent classes to be learned about.

In a decade of work as computer scientists in an industrial research setting, we’ve informally noted the frequency and value of consulting interactions. This is supported by studies of software customization behavior and spreadsheet use. Mackay [8] found that preferred strategies for getting information about software customizations were, in order: asking a person, copying and experimenting, and reading documentation. Nardi and Miller [9] uncovered extensive sharing and collaborative development of spreadsheets within organizations. And, in a rare study of experts on huge software projects, Riedl, Weitzenfeld, Freeman, Klein and Musa [11] showed that debugging expertise largely involves social and psychological expertise — their experts “preferred not to read code at all, if they could help it; certainly not to read unfamiliar code without assistance.”

Knowing that people are seen as prime sources of task-relevant information, we undertook an empirical study of the interactions between programmers learning a new system and the system’s designers. Our original goal was to study the effect of the object-oriented methodology on programmers’ problems in understanding and modifying large systems. By audiotaping interactions with the systems’ designers, we expected to focus on the apprentices’ most severe understanding problems.

However, even in the first few weeks of the study, we realized that the study provides a window on an important collaborative work practice — one which we broadly describe as an apprenticeship interaction.

Apprenticeships are common in intellectually complex domains. For example, multi-year internships are standard in medicine and clinical psychology. Teacher internship programs [4] increase the effectiveness, retention and satisfaction of the mentored. Informal, on-the-job training is also a form of apprenticeship. This is used in teaching secretaries, or librarians new to a job. In such on-the-job training, veterans help the new hires become effective by teaching a combination of facts, procedures, decision-making strategies, and the local culture. Understanding on-the-job computer training is becoming more important in society, as even blue-collar jobs become computerized [14]. Apprenticeships and consulting interactions are also common, although less formally recognized, in the computer industry. Besides the implicit references in Mackay, Nardi and Miller, and Riedl et al, a study of design meetings by Olson, Olson, Storrosten, Carter, Herbsleb and Rueter [10] describes the practice of including “greenbeans” in design meetings at one company, so they could observe experienced designers working on a real project.

Our study has provided insights into the collaborative process by which apprentices become
proficient in their task and tools, the value of the artifact-centered sessions with a mentor, and both participants’ productivity management strategies for accomplishing their individual and joint goals. We’ve begun to characterize this collaborative work practice and to identify the constraints and work styles that would need to be addressed by technological support for such interactions. Besides guiding tool development for local and remote collaboration, an understanding of the processes in effective apprenticeship interactions may eventually improve this cooperative work practice by teaching people how to be better consultants and better apprentices.

In the remainder of the paper, we first briefly describe the methodology, the complexity of the apprentices’ task, and their main strategies for making progress. Then, we introduce the range of help that mentors give, and the valuable incidental learning that occurs within the consulting sessions. With this as a background, we describe the apprentices’ strategies for minimizing their impact on the experts’ time, and the joint negotiation of the content of the sessions. We finish by discussing some implications for cooperative work technology. We suggest that technology can help facilitate cooperative apprenticeship learning by helping apprentices switch among subproblems while they wait for the consultant, by facilitating communication about the expert’s interruptibility, by making consultations more productive, and by complementing the human consultants with other computer-based aids.

2 The Study of Consulting Interactions

We conducted an exploratory observational study that focuses on consulting sessions between programmers who are learning one of two complex software systems (whom we call apprentices) and experts in the systems, who serve as consultant/mentors.¹

The three apprentices were either graduate students in computer science, working at HP Labs over the summer, or researchers within the laboratory. Each apprentice was new to the object-oriented language, C++, and to the software system. All were proficient in other high-level languages, and had been application-level programmers for over four years.

The experts were researchers at HP Labs, with around ten years programming experience. C1 was the architect/engineer of an application-builder. C2 and C3 were the architect/engineers of a hypertext toolkit. These toolkits not simple systems, but application frameworks — architectures of reusable software components, designed to fit together to facilitate building applications. Such design for reusability requires flexibility, and this contributes to complexity. Each consisted of some 200–300 C++ interrelated classes. In addition, both systems’ user interfaces were built using another C++ toolkit of more than 100 C++ classes.

Data gathering started in the first three weeks that the apprentice worked on the project and continued for 2–4 months, resulting in around eight hours of interactions on questions that nominally dealt with program understanding or object-oriented programming. To allow recording of unscheduled

¹We will often refer to these experts as consultants or mentors, since our focus is on their educational role.
interactions, informants kept tape recorders in their offices and turned them on at the start of a consultation. Taped sessions ranged from 30 seconds to over an hour long.

The audiotape transcripts provided valuable data. One can read the apprentice's initial questions, the joint discussion and troubleshooting, and incidental learning. However, to give context to the consulting sessions, we also periodically interviewed each of the apprentices, giving about six hours of interview data.\(^2\) In the interviews, apprentices described their current tasks and problems, their strategies for resolving problems before going to a consultant, their use of information sources such as source files or documentation, and the program analysis tools they wish they had. We also opportunistically collected a few data points on four other apprentices who consulted with C1 or C2, and interviewed one of those.

All methodologies have their limitations. This study did not include fleeting hallway conversations, project meetings, design discussions, or e-mail; however our audiotape-based method has the advantage of unintrusively capturing what collaborators actually do on real projects in event-driven interactions in one another's offices. Interviews showed that we obtained good coverage of system-understanding and debugging interactions. The study focuses attention on the apprenticeship interaction as an important context for complex collaborative work — the observed patterns characterize features and roles to be supported by CSCW technology.

3 The Apprentice's Task

3.1 The Complexity of 'Getting Simple Tasks Done'

The transcripts expose the range of domains that the apprentice had to learn simultaneously and the resulting complexity of the start-up process. To make changes to the software framework, apprentices had to learn everything from the details of C++ syntax, object-oriented concepts and their use, the framework’s information layout, and then of course the data structures and the dynamic information flow within the system.

In addition to this core learning, to accomplish their tasks the apprentices had to learn a host of auxiliary tools and programs. These included the compiler and linker, the file dependency maintenance program, make, various C++ program analysis and development tools, and the Emacs text editor.

The apprentices, who are competent programmers in other languages and environments, usually have the right generic strategies for accomplishing their goals. However, given this set of domains to learn simultaneously, they get tripped up by the details of programs and domains they must use.

Since they’re still learning C++ and the software framework, apprentices make many mistakes in implementing their ideas; thus, they spend most of their time tracking down framework behavior, experimenting semi-blindly, and troubleshooting problems. This experimentation leads them to

\(^2\)The interviews and consulting interactions were collected by the first author. We use the plural "we" here for expository ease.
encounter many vague or misleading error messages from debuggers, compilers, linkers, and
makefiles. As the following two examples show, the tools themselves can wind up compounding
the apprentices’ problems.

**Example 1:** Apprentices have to figure out the environment’s tools. A2 got sidetracked while
debugging a problem, when his debugging output was unexpectedly getting buffered (got printed
late). He understood the concept of buffering, figured out that that’s what was going on, but didn’t
know whether C++ or the HP-UX operating system was responsible. The concept of buffering
isn’t covered in basic books on C++ or UNIX. So, instead of working on his original problem, A2
got stuck trying to figure out different C++ printing functions, hoping that one of them wouldn’t
buffer.

**Example 2:** Uncertainty about tools’ behavior slows them down. Apprentices stay with sub­
optimal strategies since they aren’t sure of what’s possible in this toolse. Here, A1 described the
tediousness of trying different solutions to one compiler error (caused by an interaction between
her program and the software framework).

> A1: It is especially annoying in C++ because the compiler is so slow ... it doesn’t just say,
okay, she’s just changed that one little thing, I don’t have to worry about the rest of the world.
... I am using a makefile and it seems to take just as long when I do the compile, every single
time, even when I have only changed one little teeny thing. So I don’t have a good feel for
how incremental it is. ... [perhaps] it just feels very slow to a person who is used to LISP.

A1 correctly thought the makefile *should* prevent the recompilation of the whole framework.
However, to be sure there even *was* a problem, she’d have had to know how fast the C++
compiler is, how *make* really works, and the framework’s compilation dependencies. So, she was
uncomfortable, but rather than setting her original problem aside and investigating all those factors,
she put up with the slow compilation and the resulting slow experimentation. A1 described her
progress this way:

> A1: basically what it feels like to me is, I am kind of solving one little teeny problem after
another. ... It is like one pile of dirt after another. ...

When you are working with these big systems, often it is some little piecemeal thing that you
have to kind of solve before you can get beyond it and go on to the bigger problems. ... it
is often these little teeny things that can hold you up and you have to go to your information
sources to solve.

The examples show that, at least for a while, the apprentices are bombarded by ‘learning opportu­
nities’ in the guise of obstacles due to a variety of domains. However, the apprentices primarily
want to make progress on their original tasks.

How do they do this? One strategy is to ask a consultant, but these are scarce resources. As we’ll
show later, apprentices try to reduce their use of consultants and their impact on the consultants’
time. They can do this by being goal directed, reducing what they try to master, and getting by with limited knowledge. This leads to the expedient *copy and experiment* programming strategy.

### 3.2 Copy-and-Experiment Strategy

Both observations and interviews showed that apprentices recognize the overwhelming number of things to learn and consciously put off learning some of them. They minimized their learning of auxiliary tools, and even in the core system, they ignored concepts and details they didn’t feel they had to immediately understand.

As a result, a primary programming strategy became *copy and experiment*. The apprentices search their own and the system’s files for code that looks like it does something similar to what they want, copy it, and then try to figure out what to delete and what to change to make the code meet their goal.

All apprentices said that they do this to get past obstacles, to make progress without having to understand all the details. However, they were somewhat embarrassed about relying on magic. They said that such a strategy backfires; they feel they “should” understand the system and the tools better, but they believe they don’t have a choice – to progress on their primary task, they have to minimize the side trips of learning. In an interview, A1 described how (with the mentor’s help) she has a makefile that mostly works, although she doesn’t understand it; and how ambivalent she feels about putting off learning it.

> A1: I think I can patch in little things as needed, but I do find that a really confusing feature and that is an area where I probably will do programming by example and I don’t know that I will take the time, at least in the near term, to really understand [makefiles]... It’s against my religion to do that [modifying code without understanding it]. This is a violation...

> I think it is justified not to delve into that kind of thing if you can just get something that will allow you to work on the stuff that you really want to work on. Even though I am sure this policy will come back to bite me at some point... but at least I think that I can delay that moment right now.

The strategy of limiting their learning continues throughout the learning process; apprentices developed expertise in framework components central to their goal, but continue to rely on consultants for help with unfamiliar sections or interactions.

### 4 The Mentors’ Role: Not ‘Just The Facts’

The previous section hints at two of the consultants’ roles: helping the apprentices over incidental obstacles (such as C++ syntax or makefile problems), and helping the apprentices with their core framework-specific problems.
However, an interesting finding is that joint problem-solving about the apprentice’s question triggers exploration of many other topics, topics tangential to the initial question, but crucial to the apprentice’s growth into an expert. Within minutes, conversations jump around multiple domains and levels as questions are triggered by the code itself, the mentor’s explanations, and the physical actions of either participant. Discussions range from strategic, conceptual and procedural information (e.g. what is broadcasting and how does it work), to historical notes, pointers to information sources, down to syntactic details, and side-trips to discuss commands or tools that would increase the apprentice’s productivity.

The information in these discussions (a) generally is triggered by the computer-based files the pair view in order to problem solve, (b) is important to the apprentice’s growth in proficiency, and (c) focuses on information that would be hard for an apprentice to find.

**Artifacts are crucial.** Very quickly, the apprentice and consultant bring up files related to the problem — framework source files, the apprentice’s code, error output, etc. As the mentors (re)familiarize themselves with the framework code’s intent, they remember or reconstruct the design decisions, control flow, historical twists and turns, the code’s assumptions, and quirks in the implementation. If they are viewing the apprentice’s code, they may give a running commentary, pointing out errors, asking questions, and using the code to trigger discussions about algorithms, concepts, style, or auxiliary tools that would help the apprentice.

When the pair bring up a new file, the apprentice may ask about a feature brought into view — a syntactic expression, a function — or may just make an implicit request such as “this piece is confusing”. Apprentices use this as an opportunity to ask incidental questions. As mentors talk through the behavior and concepts embodied in code, apprentices’ reactions guide the conversation. They serve to confirm understanding, request clarification, or steer the conversation back to the apprentice’s immediate question.

**One question leads to another.** Not only do the artifacts serve as the source of information needed to answer the apprentice’s question, but they also serve as a starting-off point for additional learning.

Both the apprentice and the consultant take initiative in this, and negotiate about what’s worth learning, and what the apprentice wants to learn. Sometimes this is explicit, such as the following, in which A2 requests an explanation of event-handling.

A2: You say “it figures out”. Who figures out, what figures out, where?
C2: ... you don’t really have to know that.
A2: Is it the X system at some level? ... I haven’t worked with this kind of stuff before, ... in some sense I don’t really need to know a lot of detail but I just would like to have a little understanding of it.

3Note that frequently it is their own code whose intentions and flow they are reconstructing.
C2: [asks A2 to bring up another file, and, pointing at the code, describes the next layer of behavior]

Frequently, the consultant uses the task or the code to bring up relevant, interesting, or otherwise valuable information. For example, here C2 not only explained what the intent of a function is and how it interacts with other code, but also volunteered how it should have been written:

C2: ... So the whole purpose of it is... Actually, you know what should happen — I didn't think of this at the time, but now looking back, that should probably be defined to be a pure virtual function... [and goes on to explain how that would be structured]

Given the apprentices’ dominant strategy of copying and modifying code fragments, such meta-information about existing code is especially valuable.

Emphasis on “hard to find” information. We were struck by how much of the information in the sessions would have been difficult for apprentices to find written down. First, in answer to a question, mentors’ explanations tend to be much more general than what the apprentices need to fix the specific problem. Mentors focus on describing complex behavior, the rationale for a strategy, and elements of good style. This type of information is rarely documented, yet apprentices must learn this to become proficient.

An important factor of expertise is knowing about and understanding the characteristics of information sources. Not surprisingly, the mentors point these out. Again, they focus on attributes unlikely to be written down. When describing other experts, mentors and apprentices discuss how approachable other experts are and their areas of expertise. For tools, they discuss the tool’s purpose, speed, robustness, etc. For written information sources, they describe whether the source is accurate, readable, up-to-date, and where it is located. For example:

C1: [that's a] Question that [the local C++ expert] could answer ... Or, you could go and read the ‘Annotated Reference Manual’.
A1: What’s that?
C1: The Annotated Reference Manual is ... [ it focuses on] explaining why things are done the way they are. In fact, it is the only positively authoritative definition. The 2.0 notes that came with C++ are inaccurate.

4.1 Interactions Provide Crucial “Incidental Learning”

The transcripts vividly show that interactions with the expert are much richer than a simple answer to the apprentice’s question. A major aspect is the incidental learning we see above — triggered by the the question or the problem-solving, but about facts, domains, history, assumptions, strategies beyond the apprentice’s original question.
Apprentices learn incidental information from what the consultant volunteers, from questions they ask opportunistically, and from observing the consultant’s troubleshooting strategies and tool use. As the above examples show, incidental doesn’t mean unimportant. We believe this represents a valuable mechanism by which the programmers become proficient in a new environment, and that this incidental learning in fact represents a major benefit of the interaction.

5 Balancing Apprentice and Mentor Productivity

To simultaneously learn a language, software system, and toolset is a prolonged task. As we’ve seen, experts significantly reduce the apprentice’s learning difficulties. They help apprentices localize and solve problems, understand and apply concepts, and they critique programmers’ partial solutions, thus helping avoid time-consuming detours.

However, mentors are a scarce resource; they aren’t always available. Also, both parties are aware that mentors have multiple apprentices asking them for help, and that they have their own work to do. The mentors may even be working on a module which the apprentice needs for another subproblem.

Since our study focused on apprentices’ questions and use of information, our primary data is on the apprentices’ strategies. We have shown how apprentices use the copy and modify coding strategy to get answers without going to the expert. Here we discuss apprentices strategies for reducing their impact on the mentors, and for making effective use of the interaction time.

5.1 Minimizing the Impact on the Expert’s Time

The apprentices emphasized that the consultants were easy to approach, interruptible, and friendly. However, it was clear that apprentices explicitly work to minimize their impact on the expert’s time.

First, they spend up front time to try to understand or narrow the problem before going to the consultant. This is visible in the sessions, where the apprentices guide the consultant to look at specific files, describe their hypotheses, or summarize what they’ve already tried to do. Also, as consultants scan a source file and comment on the code, the apprentices interject to show they understand a segment, that they have figured out a connection, or that that’s where they’re confused.

How long they wait before approaching the expert depends on many factors. These include the urgency of the problem (whether there’s something else they can be productive on), the perceived length of the interruption, the apprentice’s personal style, their view of the consultant’s style, the consultant’s proximity, and as one apprentice said, “how lazy I’m feeling”.

Apprentices seem to have one of two preferred communication styles — either quickly walking over, or spending more time trying to puzzle the problem out, and then send e-mail. Generally all agreed that for non-trivial problems, interactivity and being able to see context on the screens is
crucial. Apprentices who send e-mail do it to be less intrusive. The ones who walk over emphasize the benefit of maintaining momentum and being able to discuss the problem interactively, in the context of the relevant artifacts. Both types recognized the value of the other style, but found it difficult to adopt.

Second, they multi-task. They switch from one sub-problem to another as they reach obstacles. This context switching has multiple causes. Sometimes, the apprentice and consultant work different hours — A1 was on an early schedule, while C1 tended to work late; another consultant sometimes worked from home, and two apprentices with looming deadlines worked late into the night. Thus, apprentices sent e-mail to the consultants and went on to another subtask.

An effect of task switching is that one can easily build a stack of subproblems. Thus, when the two meet, the apprentice may have a set of questions about a variety of artifacts and domains. A1, who accumulates questions, has developed strategies to reduce forgetting. She keeps a list of questions in a file and keeps a project notebook in which she writes notes during her meetings with the consultant. After each meeting, she marks the action items. Thus, even though programming is only one of about three major tasks for her, the notebook helps in “swapping everything back in and starting up and just going through and doing the action items”.

5.2 Negotiating the Conversation's Content

The conversation’s content is implicitly negotiated between the apprentice and the consultant. Sometimes, as we’ve seen, the apprentice asks for clarification about a term or why something happens. Often the consultant volunteers information about a concept, or some historical information on code they’re skimming (e.g. how it should have been written, or that the code will go away in the future). Or, a discussion about an error leads to a discussion about C++ style or the system’s design.

The breadth and depth of the discussion is negotiated between the two. Each person frequently restates what they’re hearing or clarifies a question. This often serves as a trigger to raise or lower the level of abstraction. If the apprentices don’t understand a concept and care about it, they request clarification. On the other hand, mentors frequently volunteer information they believe the apprentices should know. The apprentices don’t always want to learn the theory, or can’t yet follow the detailed rationale and control flow. Thus, they bring the conversation back to the problem at hand, with statements such as “pop up a couple of levels, please”. Experts sometimes explicitly reduce the scope by saying things like “we don’t have to worry about that. I don’t even know what code decides that.”.

In general, the apprentices know better what they want out of the interaction, but the experts introduce what they believe should be relevant to the apprentice. Apprentices guide the flow of the conversation — switching to a new question once they’re satisfied, interjecting with a side question on a function on the screen, asking for additional explanations, or asking for more details of how to proceed next. So, the overall pattern seems to be that the apprentice starts the conversation,
the consultant brings up relevant (or incidental) information, and the apprentice then guides the
conversation to get more breadth, depth, or to follow a random useful side track.

6 How Can CSCW Tools Help?

Computer support for the cooperative learning can take a number of forms. Here we focus on three:
support for encapsulating the information related to a problem, support for remote consultations
and the negotiation of availability, and support for getting answers from a distribution list or an
archive of previously asked questions.

6.1 Encapsulating Context and Task-Switching

Problems whose source isn’t obvious, and more general questions, like ‘how should I write this?’,
or ‘how does this work?’ require examining examples and data from many sources. Thus, when
the apprentice and consultant meet after the apprentice has gone on to another problem, it would
be useful to be able to bring up the problem, its working set of files, plus notes of the apprentice’s
specific questions and hypotheses raised by the files.

Support for task-switching could also help in the high-pressure situations near a deadline. At those
times, everyone tries to make progress on multiple threads. Programmers switch among multiple
separate subproblems on an interrupt-driven basis in order to make progress while waiting for a
response from someone else, to respond to another’s pressing problem, or to figure out the source
of an error.

Encapsulating a set of files and the apprentice’s local questions is quite doable. It requires
separating files and programs spatially into workspaces, allowing elements to reside in multiple
workspaces. These workspaces then form task-based contexts that the user may switch between
[6]. Since different solutions often require incompatible changes to multiple files, pathnames, etc.
a more interesting extension of current workspaces is one that would support cloning a context
and simultaneously trying out divergent solution paths – with separate file versions, environment
variables, etc. in each context.

6.2 Facilitating Remote Consultations

All subjects said they’d still walk over to talk to someone physically separated by most of a building
or a floor, because there is such great utility in seeing the same screens, drawing on a whiteboard,
or together moving around files, directories, or modifying and testing code.

In situations where walking over is impractical, props for high-context collaboration have to be
provided [3]. Some aids already exist for sharing windows and pointers across a network, including
HP’s X Window System extension, HP SharedX, prototype shared editing tools such as GROVE [5]
and ShrEdit [13], and shared drawing tools such as TeamWorkStation [7], or VideoDraw [15].
the shared work-surfaces’ performance and the audio and video quality are good, such tools may allow high-quality collaborative work.

A major problem of remote collaborations is deciding whether and when to interrupt. From a group productivity perspective, if the apprentice simply needs an answer to a quick question or a confirmation that they’re on the right path, it often makes sense to interrupt. When the pair are nearby or in the same room, there’s low-cost or implicit signaling of needs and availability, allowing for more frequent but less-intrusive interruptions. As Root [12] points out in discussing Cruiser, people use a host of audio-visual clues to determine another’s availability.

In distributed teams, or in organizations in which many tools and systems are used simultaneously, experts are often distant from other users. Thus we need aids that provide similar mutual awareness, access to one another’s computers, and low cost interruptions as are available in a shared office, but that make possible the privacy and concentration time of individual offices.

6.3 Other Collaborative Learning Aids

Besides facilitating task-switching and communication with a consultant, computers can help the apprentice get information from a broader community of programmers, or tap into knowledge from others’ previous questions.

On the Unix system, there are electronic news groups organized around a software system (e.g. Andrew, the X Window System). There, experts and novices discuss bugs, features, and give examples of how to do things. Thus, in a larger community, apprentices can learn from each other as well as from experts. Late-comers to the notes can either ask questions, or they can browse the already collected information. Apprentices may browse the archive of the discussions to pick up terminology, concepts, techniques, and query it to get an answer to a previously asked question. In some news groups, a few experts collect and periodically post “Frequently Asked Questions” (FAQ). These experts donate their time to the maintenance and periodic posting of the FAQ. This heads off many novice questions, and the experts avoid repeated collection and (re)validation of information.

A variant on this informal mechanism is Answer Garden [1], an e-mail based system with a growing tree structure of diagnostic questions and answers. If the learner’s question isn’t answered within the system, the diagnostic questions are used to identify the appropriate expert to whom the question should be directed.

Since, consultants often use interactive discussion or other context on the apprentices’ screens to clarify the problem or its cause, an e-mail forum is not as effective at troubleshooting as a face to face session. Even more serious is the fact that these mechanisms do not provide the high-bandwidth communication and the opportunities for incidental learning that our apprentices had. Thus, even in the e-mail interaction, apprentices are more likely to get just the answer they asked for, without the richness and breadth seen in our consulting interactions.
However, the archive of previously asked questions, backed by access to users makes such captured expertise very important, especially in contexts where local experts are unavailable or where there is a large stream of mostly repetitive questions.

7 Conclusions

Apprenticeship learning is a collaborative work practice of theoretical and practical significance. It is a common context by which real problems get solved and learners pick up the range of knowledge needed to become experts. Apprenticeship should be especially interesting to the CSCW community because of its characteristics: a focus on intellectually complex activities, cooperative problem-solving with shared computer artifacts, a very fluid negotiation of content, and a rich shared context that enables both task-oriented and incidental teaching. Also, the apprentices’ desire to reduce their impact on the consultants’ time impacts their work pattern and interaction style in ways relevant for technology design.

This study characterizes the consulting interaction in complex, computer-supported domains, and exposes issues to be addressed by technologies for supporting collaborative work. The interaction pattern we’ve seen may generalize far beyond computer programmers to other workers in many complex computer-based domains — from reference librarians to engineer CAD users to spreadsheet users — in which employees must accomplish complex tasks with little training. Apprenticeships are costly in terms of the expert’s time. However, we believe that the rich transmission of expertise which this study has shown makes them appropriate in many domains — domains where there is an immense amount to learn, where there isn’t formal support, where people have good working relationships with colleagues [2], and where the goal is for everyone to become proficient in the toolset.

We have confirmed that computer-based tasks require knowing details about many tools and concepts, often details conceptually peripheral to the goals of the apprentices. We found consulting interactions to involve a complex negotiation of the content, with the apprentice in charge of the overall direction. The shared artifacts and cooperative problem-solving trigger much incidental learning, learning necessary for the apprentices to grow into experts. Thus, we believe that consulting interactions are crucial, not only in helping the learners past immediate obstacles, but in transmitting a variety of useful, hard-to-find information essential to proficiency.

Because technology users are often not co-located with experts, CSCW tools are needed to make remote interactions effective. We’ve suggested that tools for consulting must address multiple aspects of the work situation: enabling apprentices to encapsulate information as they prepare for a meeting, enabling apprentices to switch tasks, facilitating negotiation about the consultant’s availability, facilitating the interaction itself, and extending the apprentice’s community of consultants via electronic bulletin boards and other information sharing tools. Given the richness and task-appropriateness of the learning in the consulting interactions, this is an unparalleled educational mechanism, one that should be explicitly supported by both organizational structures and by cooperative work technology.
8 Acknowledgements

Many thanks to Mike Creech, Allan Kuchinsky, Bonnie Nardi, Andreas Paepcke, and Craig Zarmer for their helpful comments on drafts of this paper. We are grateful to the informants who gave their time to this study, who took care to turn on the tape recorders, and who shared their travails and their insights with us.

9 References


