Beyond Program Understanding: A Look at Programming Expertise in Industry

Lucy Berlin
Software Technology Laboratory
HPL-92-142
November, 1992

In the computer industry, expert programmers must often relearn parts of their craft as they retool themselves to new computer languages, programming environments, software frameworks and systems. Our study of consulting interactions between these apprentices and experts has given insights into this collaborative work practice and into the knowledge gaps of programmers in a new environment.

In this paper we characterize the apprenticeship interactions we observed, the skills experts use in collaborative problem solving, the hard-to-find information they emphasize, and the tutoring skills they exhibit. The observations also indirectly suggest the multi-faceted knowledge required for real-life programming expertise, and the knowledge and skills that make experts so much more effective in their daily work.
1 Introduction

As pointed out by T.R.G. Green, "programming is an exceedingly diverse activity" [8]. It is traditionally broken down into subtasks such as problem understanding, program design, coding, and debugging, [10]. Students are taught programming languages, principles of subsystems (databases, operating systems) and the concepts and principles of design – concepts such as data abstraction and object-oriented design. However, for professional programmers, true expertise involves much more than knowledge of those concepts — it often involves knowledge of an application domain (chemistry, accounting, etc), knowledge of a particular system that one is working with, and proficiency in the languages and tools of one's environment.

To companies, programming expertise and its development are extremely important issues. Given the dramatic differences in individual productivity among professional programmers [14, 5], companies have much to gain from raising the average programmers' skills. In addition, expertise is not permanent. Programmers who switch jobs or even projects within a company often wind up learning a new application domain, a new software system, and frequently a new language and toolset. Thus, as programmers switch projects, much of their specific knowledge becomes inapplicable and they become informal apprentices to the experts in the new environment.

Thus, programming experts in industry have two intertwined roles. They are simultaneously:

- **software engineers** who develop functionality, and
- **mentors or information resources** to those who know less

This paper informally explores both factors of programming expertise. It reports on an exploratory study of consulting interactions between professional programmers who are retooling themselves, and the experts who serve as their informal mentors. The initial goal of the study was to show the effects of an object-oriented methodology on programmers' problems in understanding and modifying large systems. However, one of the study's major benefits was to expose the rich learning that goes on within these interactions, showing that the consultations represent an informal *apprenticeship*. We report on the collaborative work aspects of this apprenticeship in [3], where we describe the apprentices' strategies, characterize the collaborative interaction and suggest how computer-based tools might facilitate this type of cooperative learning.

In this paper we focus on the expert. Our data exposes the complex ways in which experts help the apprentices, and the range of domains that experts must span. The interactions also allow us to conjecture about some differences between the apprentices and experts and how those differences enhance the experts' productivity.

2 Methodology

We conducted an exploratory observational study of interactions between professional programmers learning a new system and the system's designers. The data consisted of two types of verbal
protocols: constructive interactions between the pairs and interviews. The participants were six professional programmers: three experts and three apprentices.

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. The toolkits were not simple systems: they were complex application frameworks — architectures of reusable software components, designed to fit together to facilitate building applications. Each consisted of some 200-300 C++ classes. In addition, both systems’ user interfaces were built using the InterViews [9] toolkit, a toolkit of more than 100 C++ classes. Object-oriented design results in control flow that is distributed among many classes, and the goal of reusability also yields additional complexity in order to provide hooks for flexibility [2]. The hypertext system was implemented as a distributed system composed of multiple event-driven interacting processes; such a design greatly distributes the flow of control.

The apprentices were graduate students or professional researchers. 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 apprentices’ tasks were to extend the toolkits by adding a subsystem or a fundamental new capability.

Data gathering started in the first three weeks that the apprentice worked on the project and continued for 2—4 months, resulting in a total of around eight hours of transcribed verbal interactions on questions that nominally dealt with program understanding or object-oriented programming. To allow data gathering of impromptu consultations, informants kept tape recorders in their offices and turned them on at the start of consultations. For practical reasons, we did not include fleeting hallway conversations, project meetings, design discussions, or e-mail. This audiotaping technique allowed us to unintrusively capture naturalistic data on expert and apprentice work interactions.

To give context to the consulting sessions, we also periodically interviewed each of the apprentices and one expert, audiotaping most of the sessions and yielding some six hours of audiotaped interview data. 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. The interviews showed that we obtained good coverage of system-understanding and debugging interactions, only limited samples of design sessions, and no direct data on software integration and joint system testing sessions.

3 Results

Our data suggests that the apprentices represent an interesting niche in proficiency, with a different combination of skills and problems than normally studied. Using excerpted quotes of interactions and interviews, we expose the complex ways in which experts help these apprentices and the range of domains that experts must span. The interactions also allow us to conjecture about the differences between the apprentices and experts.
3.1 Expert Apprentices — Neither Novices nor Experts

The apprentices were experienced programmers retooling themselves to a new system, language and toolset. As experienced programmers they generally had reasonable generic strategies for their tasks. They understood general programming-related operations such as editing and compilation, and advanced programming concepts such as output buffering, dynamic arrays, garbage collection, class initialization and event handling. They had sensible generic debugging strategies for programming – strategies such as writing small test files, adding print statements, examining stack traces and searching for function and variable definitions.

Since they had all this experience, these programmers were expected to learn what they needed to know while getting an assigned task done. As in the workplace training described by Scribner and Sachs [13], their cross-training occurred within the context of work tasks, with exposition and practice together. However, unlike Scribner’s stockroom trainees, in our context the programmers did not begin their retraining by shadowing a trainer. They immediately began working independently. They designed and implemented new code, learning from existing code, from manuals and from the people around them.

As described in Berlin and Jeffries [3], the apprentices’ primary strategy for getting work done was to copy and experiment with existing code. This ubiquitous copy-and-modify strategy, also observed by Visser [15] and by Flor [6] meant that the learners did many things they didn’t understand. They often got unexpected results, or the code simply broke. Apprentices then went to the experts to find out “why do neither of these work?” or “when I made this modification, I didn’t expect ...”, “what does this mean” or “what am I missing [in my reading of your code]”. Sometimes they showed the expert some code and asked for validation, as in “is this going to work?”

Early on, one apprentice, competent in object-oriented programming in Lisp/CLOS, noted regretfully that her current problems are “little and uninteresting and kind of trivial and piecemeal”. In the initial month [of part time work] she’d designed and wrote skeleton subclasses of two system classes, learning the essentials of C++ and the toolkit. However, even after her prototype compiled stand-alone, she spent a couple more weeks dealing with obstacles such as makefiles, network file system mounting, and conflicts between her own variables and global variable declarations in the system toolkit, and additional C++ misunderstandings. She had the right design strategies and was able to express a detailed design, but in the initial months she was constantly blocked by her ignorance of the pragmatic details of C++, the system design and the classes she was subclassing, and the tools and quirks of her software environment.

The mix of problems apprentices brought to the experts changed over time. As the apprentices learned the language, overcame the initial set-up hurdles and learned the common error messages, more of their questions for the experts dealt with with the toolkits, and with problems in understanding, using and extending the toolkit’s functionality. However, throughout the 4-month study, the toolkit and design questions remained interspersed with problems such as how to express specific concepts in C++, problems with unix, with ancillary tools such as xdb++ or X Windows, and with unclear error messages.
4 Experts as Mentors

Even though the apprentice starts by asking about a single issue or problem, the resulting interaction is striking in its richness. Even though the experts had no training in coaching, they seem to understand their pedagogical role. Not only does the expert provide the answer to the explicit question, but the apprentice gets a waterfall of valuable information. Let us examine some important characteristics of the expert's explanations: its multi-faceted nature, the process of reconstructing design from code, their emphasis on hard-to-find information, and the highly collaborative nature of the conversation.

4.1 Experts answer much more than the initial question

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. These topics sometimes are tangential to the apprentice's initial question, but they are crucial to the learner's growth into an expert.

Once the apprentices no longer spend most of their time struggling with the language and ancillary tools, much of the discussion revolves around the system's design rationale, goals, and tradeoffs. Experts give the rationale for design decisions and fill in information about how entrenched that decision is - how many classes (modules) would have to change in the system if that decision were changed. For example, here the apprentice wanted class X to know about its related window class Y:

**Expert1**: The reason [component-class X] doesn't know about [window-class Y], and the reason why [X] is different from [class Z], is to allow [X] to be implemented without using [Y]. The [class AA] is the first and only example of this usage, but the principle seems important enough to preserve. At least, as long as we live in a world where components are windows and windows are expensive.

Generally such an answer is followed by a discussion of the system's principles and the system implementation. If the existing system design does not meet the apprentice's needs, they discuss alternate solutions that and evaluate if those preserve the driving principles and avoid major rework. As shown above, the expert supplies the details of system's interactions, recalls where the functionality is used, reconstructs a plausible design rationale, and evaluates the importance of the design principles.
4.2 Experts reconstruct the design and design rationale from the code

In such detailed explanations, being able to see the code is crucial. The expert uses the code, debugging traces and the documentation as clues for reconstructing what is going on and why. Sometimes this is code he himself had written within the last year or two, sometimes it is someone else’s code. As is shown below, once the code or debugger trace are visible, the code’s flow, its intentions, and the relevance to the current problem are quickly reconstructed.

**Expert2:** We need to look at what this [method] does. Does this eventually call this one, or is it completely rewritten? ... [the apprentice fills in what happens] ... Right, it will get this one, right. ... right; that’s why it has an argument...

Especially in object-oriented code, the behavior defined in one class is not the full behavior, and abstract classes may have code that is intended as a hook for further specialization. The experts are more likely to notice clues to look elsewhere, even when reading someone else’s code:

**Appr2:** There. ... Now if you go look at this hyperupdate, it is a no-op.

**Expert2:** [FOO-object], sure, that makes sense. Does that not make sense? ... All that is happening is, Let’s see, [finds the function definition and summarizes its control flow]

**Appr2:** So it always calls this, and it’s always a no-op.

**Expert2:** No it’s not. Right. This is what you are missing. In [FOO-object], what you have to ask is, [and finds the FOO-object file] Right there. Hyperupdate. Virtual. Okay. So the whole purpose of it is, [and explains why the parent class’s method is defined as a virtual] ... So he inherits from [FOO-object] which means he gets a method called hyperupdate ... If I didn’t do that, then I can’t guarantee ... that there is a hyperupdate method on all these different objects that he has and the objects I have so they all inherit from FOO-object.

Thus, the expertise here lies not in having memorized the system, but in checking out hypotheses that seem odd, and in being able to reconstruct or the design and rationale in reading distributed code. In the second example, the expert first agrees that an unfamiliar method doesn’t have any effect [is a no-op], then he looks at the actual code and corrects himself, and explains why the code was written the way it was.

4.3 Experts emphasize hard-to-find information

We were struck by how much of the information would have been difficult for the apprentices to acquire on their own. Apprentices have access to the code, but it is the experts that describe the unwritten design: the methods’ purpose, the interactions among classes and methods, and the rules for when parameters expect explicit values.
Experts’ explanations also focus on concepts and explanations of complex behavior, rationale for a strategy. In the course of answering an initial question, they point out good and bad examples of code — information that is priceless given the apprentice’s dominant strategy of copy and modifying code. Experts also suggest and describe information sources — other users, advanced manuals, guides to C++ error messages, source code libraries, etc. There they focus on attributes unlikely to be written down. They describe other people’s areas of expertise and their approachability. For new tools, they describe not just how to use it, but the purpose, speed, and robustness. They point out documentation and its reliability, location, and tricks of use.

This type of information is rarely documented, yet is essential for proficiency. It is essential because it gives the apprentices additional problem-solving schemas, suggests rules for choosing a plan, or gives rules for understanding code whose use is ambiguous.

For example, here an apprentice had used the copy and modify strategy to create a method definition with a default parameter, but she really didn’t know how to use the code she’d copied. The expert gives a running commentary on her design while reading the code to answer the original question:

**Expert1:** You’ve defined [an initialization method] correctly to take a default value.

**Appr1:** Right.

**Expert1:** So that way, when somebody invokes, when somebody makes an instance of [your class], they should never pass an explicit value to that.

**Appr1:** They shouldn’t?

**Expert1:** They shouldn’t. [explains the normal data flow] and this is all the game that is being played to try and make [method] `initialize` be run [exactly] once. And if somebody were to pass an explicit value of [the parameter], then `initialize` would not run at all.

When considering using tools and information sources, the experts check whether the apprentice is familiar with it, e.g. “did you ever get [the static analyzer]?”, or they may explain the use and drawbacks of a book:

**expert2:** Actually, you know what we can do is to look up [the number of the warning message]... do you have this [guide to C++ error messages]?

**A2:** No, not yet.

**expert:** Let’s see if we can find this. It is number 123. These error messages don’t quite map, this is not totally up to date ...

Such a running commentary about the tools and information sources, peripheral problems, and about what the expert was noticing or doing was ubiquitous. All the experts talked aloud as they read code or considered hypotheses, even though none had a training background. This
strategy turns out multiple benefits. First, it unintrusively demonstrates the expert's problem-solving strategies. Second, it involves the apprentice in the problem-solving process, allowing the apprentice to interject hypotheses and questions about what's being said, and information on files the apprentice may have examined when trying to solve out the problem alone. It also contributes to a very interactive learning style, in which both the apprentice and the expert negotiate what will be talked about.

4.4 Experts and apprentices use a highly collaborative conversation style

In answering the learner's question, experts also give pointers on skills necessary for general proficiency in the language and environment. This includes programming style suggestions; useful unix, editing, or X-Window manager commands; debugging tools and strategies; and pointers to valuable information sources. When explaining the system's actions, they often pop up a level to explain the general programming concept (e.g., of class constructors, virtual functions, polymorphism, or forward references), and then back down to show how it applies to the system's code.

The flow of the conversation is implicitly negotiated by both the expert and apprentice. Experts volunteer information they think the apprentice should know — things such as the general concepts or the productivity tricks. Both also follow tangents triggered by the problem solving discussion or by code visible on the screen, (e.g., "so what is this"); or "oo, this is interesting", or "wait, wait. I just saw something that looked wrong".

Unlike lectures, these conversations are very collaborative. Experts invite apprentices' interjections by pauses and by comments such as "right?" or "well that's interesting...", to which apprentices may respond by asking a question or volunteering what they know. Apprentices signal their level of understanding by utterances such as "hmm?", or by questions that restate the solution, e.g., "you mean this unique-ID thing?", or "so this is in the browser"? Often they interrupt to help explain the solution — a conversational technique that serves two purposes. It shows what they understand, e.g., "Right, [that's true] for any annotation.", and it implicitly asks for validation or help: "Okay, it will work with legal-class. It will test to see — what is the test it is doing?". This collaborative process of conversational grounding, described by Clark and Brennan [4], is important to successful explanations. Because the pair continually seek and provide evidence that each understands the other, the participants quickly repair misunderstandings or confirm that there is mutual understanding.

The collaborative style is also similar to the interactional processes described in the empirical work of Fox [7] on tutoring assistance. Fox concludes that strategies that characterize tutoring — questioning inflections, pauses, collaborative completion — enable tutors to check on the learners' understanding and facilitate successful learning while minimizing overt corrections.

In sum, we believe that the mentoring skills of experts lie not only in their knowledge of the programming domains and in highlighting what the apprentice needs to learn, but also in the collaborative conversational skills of grounding and of tutoring dialogue.
What Comprises Programming Expertise?

The previous section focused on the behaviors and interactional skill that make programming experts valuable as mentors to apprentices. The interviews and transcripts of the interactions also indirectly shed light on the knowledge required for programming expertise, and on the skills that make experts so much more effective in their daily work.

5.1 Programming is a multi-faceted craft

The first striking result is the range of separate domains and the mass of facts within each domain that an expert spans in order to be productive. This breadth is visible in the range of information that experts bring to the interactions — information ranging from application philosophy and requirements, system design and control flow to historical and design quirks. Beyond this knowledge of the application’s domain and half dozen subsystems that make up the application, the expert must be proficient in over a dozen non-intuitive and idiosyncratic ancillary tools such as the compiler, linker, multiple debuggers, editor, makefile, unix shells, etc. These tools each have their own syntax, obscure error messages, and their effects interact in complex ways.

As an example, here is one expert’s enumeration of the hurdles faced by A2, a very competent C programmer who, essentially as soon as he arrived, had to start building extensions to the hypertext prototype:

**Expert2:** A2 didn’t know C++, wasn’t familiar with hypertext systems [the programming sub-domain], didn’t know [the experimental system], didn’t know Lisp [another language]. Most of the things he worked on he didn’t have any real knowledge about...I think there was a fair amount of gear-up time in just trying to learn what C++ [syntax] means...they are very cryptic and we don’t have any decent information to read about it either.... you have all these weird forms like CONS CHAR*, CHAR CONS*, you can put the constant in forms and it has a somewhat different meaning...

He did a lot of shell script programming, which he hadn’t done a whole lot of, so he needed to look up a whole lot of things with that, so it was just a very big learning curve.

... he spent a lot of time working on [a distributed system communication module], which had no documentation and its interface was changing and only [the module’s author] knew how it worked....

And [he lacked] understanding object oriented programming....understanding exactly what happens for constructors and destructors in terms of what’s the ordering; the things that occur when using inheritance. Things like that are kind of dynamic and they’re difficult to understand even if they are written up. [And as a result of the inheritance]... there was a little bit of difficulty with needing to go look somewhere else because [classes] are distributedly defined.

He would [also] not use [the Emacs text editor] very effectively. He didn’t know [Emacs] things which would help format and understand C++ code, and the modes
Apprentice A1 had as many problems, even though her background and skills were completely different. She had done significant object-oriented programming in a Lisp-based language and was a power user of the Emacs editor, but she wasn’t at all familiar with the compiled language C or its object-oriented derivative language C++. Thus, she too was trying to build extensions to the system while learning the system and the C++ language, but she had the additional handicap of simultaneously having to learn the concepts and syntactic quirks of the base language C, C compiler messages and C-based development tools such as makefiles, the linker, and the cdb debugger. Since textbooks of C++ are written for C programmers, they all assumed knowledge she did not possess – knowledge of C syntax, terminology, concepts and ancillary tools.

Over a period of months the apprentices became proficient with relevant sections of the system, the new programming languages and the basics of the ancillary tools. Actually, one should say that they were semi-proficient in the routine activities, and expert in a working-set of modules of the system. What differentiated them from the true experts was their depth and breadth of knowledge – their uneven knowledge of the system and the ancillary tools.

5.2 Knowledge is Speed

Looking back over a four-month project, one expert described his impression of the things that were most time consuming for the apprentice:

**Expert2:** [as an example] It took us way, way, longer and [it took Appr2] way, way, longer than expected to get [a system extension] working. We [first] attempted to do it with scripts... [We] got in over our heads because we didn’t realize the multi-user contention problems we’d run into... We both got this thing barely running — working long hours; new for me becoming more of a shell-script hacker, and it was terrible, it was a complete mess and it didn’t work. ... [so] then we started from ground zero and learned [an experimental system for distributed multi-process communication] and everything else and got to the point that we could use it.

System kinds of things, low level, are just incredibly time consuming, and those can be moving files around, lots of management of where do things go, housekeeping, connecting up with machines, getting files, all of that kind of stuff. Then writing little scripts to do things which seem like they kind of work 80% of the time, but then the other 20% they don’t, which cause all kinds of problems.

It’s that kind of stuff that ate up all the time. It wasn’t nice, high-level questions like "How do you structure this?" or "What’s this mean in the C++ code?"

Thus the major obstacles to productivity may be time to learn new skills, the laboriously carved false paths, and the myriad of small details that must be managed in system development. Because apprentices are less familiar with all the knowledge domains in the programming environment,
each task takes disproportionately longer. It takes them longer to correctly deal with the small
details and to express and test their designs enough to recover from false paths.

5.3 Why are experts more productive?

Given the diversity of domains to learn and the complexity of the environment the programmers
are dealing with, we can conjecture about some interesting productivity differences between the
experts and the apprentices, and the causes of the differences.

Chi, Glaser and Farr [1] explain experts’ speed in two ways. They suggest that in cognitive tasks,
experts “arrive at a solution without conducting extensive search”. In routine tasks [such as typing]
the speed is acquired over many hours of practice.

In this study, we do not have any measurements of programming productivity of our apprentices, or
of the experts. However, the interactions between experts and apprentices suggest three differences
that have a major impact on productivity. These differences are interesting for they have little to
do with competence in the general skills of program design, coding or debugging.

1. Experts are less derailed by minor obstacles.

Much of programming involves semi-routine subtasks. Prototyping an idea or testing a hypothesis
requires the ability to find and interpret the appropriate toolkit code, express a concept in the
language, and recover from the minor errors one introduces. Each of these auxiliary subproblems
turned into error-prone problem-solving activities for the apprentices.

Experts not only make fewer errors in the semi-routine subtasks, they are also less derailed by
errors. We saw that experts are less dependent on accurate clues to help them recover from the
errors. Apprentices are both less certain of what they know and less likely to recognize disguised
clues to problems. Thus, for each minor problem, they have a broader problem space to search,
are slower at searching it, and they are more misled by inaccurate messages. Experts have many
specific condition-action rules, such as “If a makefile’s line looks correct, look for spaces where
there should be a tab”. These allow them to compensate for unhelpful messages such as “syntax
error” and even for semantically misleading clues.

2. Experts use power tools to facilitate common subtasks.

   Expert1: I have an alias, triple-x or something which is easy to type which says,
   “run the debugger with these directories”

The programming environment requires that practitioners do many tasks repetitively. These in­clude compiling a series of files, making an almost identical change to many files, checking for
mismatched brackets in code, or invoking long complex pathnames or commands and their options.

Experts often simplify such subtasks by using productivity tools in their environment. They use
editing macros to automate repeated editing operations, editing modes to help check for mismatched
brackets, *makefiles* that automate compilation, shell *aliases* and shell *scripts* to simplify complex command invocations, *window manager* customizations to help them manage the dozens of files they work with, etc.

We found that the apprentices do fewer such simplifications, except with the few tools they were already proficient in. They aren’t sure how long a side-track would be required to figure out and implement any ‘productivity enhancement’. Thus, even when the experts suggest the tools, apprentices often put off the learning required tools, or they try and often abandon the attempt before succeeding. As a result of their lack of skill with the environment’s power tools, the apprentices have more complex plans for their common subtasks. In addition, each step may take more effort to remember, and will be more error-prone.

Experts also use more productivity tools in debugging and program analysis. Experts suggested and sometimes demonstrated debugging tools to the apprentices, (e.g. “Well, use the make file or give the full incantation to CC. Your choice, but you will be happier with the make file.”) Apprentices often put off learning the tools, or tried and rejected the tools if they couldn’t successfully use them on the first problems they attempted. In contrast, each expert comfortably used multiple different debugging tools, which facilitated their debugging—they were willing to construct precise execution breakpoints, interpret stack traces and examine core dump files.

3. *Experts use other experts faster*

Experts quickly refer to other human experts for their hard problems. Chi et al. [1] suggest that that may be due to better self-monitoring skills that make them aware when they make errors and why they fail to comprehend. We suggest that social/cultural factors are crucial in delaying apprentices’ use of experts.

Both experts and apprentices get into situations when they need help. However, the data suggest that experts may ask for help more often when they need it. Our apprentices do know when they fail to comprehend, but they feel obliged to struggle through or learn what they can before “bothering” the expert. This is not invariably true; the time spent before asking seems to depend on the individuals’ immediate mood, their general interaction style and their physical and organizational proximity to the experts.

Here are sample quotes that show the difference between the experts and the apprentices’ descriptions of their actions:

**Appr1**: My policy is usually to try to find something on my own, just for the exercise of doing it and to avoid overloading [my mentor]. If I am really frustrated, if I have been looking and looking and can’t find it, like what happened with the *makefile* where I did as much as I could and [when] I just knew I was not going to make any progress without help, then of course I would go and ask.

**Appr3**: I don’t go till I can ask the question in terms that are reasonable... until I’ve done some exploring, so I go hunting around to figure out what’s going on. ... So,
I don’t ask till one of two things happens – unless either my back is against the wall, and then I describe what I know, or until I come to a genuine area of understanding [of what’s wrong, and can describe it more compactly]

Expert 2: I did a lot of walking. When we were doing the port [to machines X and Y], I needed to ask a lot of people questions anyway, who were experts about [X], like [one expert] or knowledgeable about the [Y] port; InterViews, [expert1 and expert3]. So I was constantly running around and we’d make a little progress and find the problem and try to figure it out.

Why might experts feel freer to ask? First, as the quotes show, learners expect themselves to try figure things out on their own. The culture also encourages primarily semantic questions, not questions about “simple” technical problems with the environment, productivity tools or syntax. For example, Riecken, Koenemann-Belliveau and Robertson [11] point out out that in their culture, “expert programmers consider comprehension of sophisticated coding techniques a ‘rite of passage’” for the learners. Second, as Riedel et. al., [12] point out, experts have a more reciprocal relationship with one another, and this makes it easier to ask for help. In contrast apprentices not only do not have a reciprocal relationship, they have more questions and problems and thus feel that they are more likely to overuse the expert.

6 Summary

This paper has taken initial steps toward recognizing and understanding the environment-specific aspects of programming expertise and the valuable role of consulting interactions in the development of that expertise. We have described how competent programmers become semi-novices as they retool themselves to a new language, application development framework, and the set of ancillary tools in their new programming environment. For a period of months, the programmers’ generic skills in programming design, comprehension and debugging are made ineffective. The programmers’ attempts are thwarted by their lack of facility with the concepts and details of their new language, the system and the myriad of ancillary programs.

Our exploratory study of consulting interactions between these programmers and the experts on their new projects has shed light on how the programmers develop proficiency in their new environment. We argue that the experts serve as mentors in an informal apprenticeship. We believe that the 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.

Within the consulting interactions, the apprentices’ initial questions serve as a jumping-off point for a widely ranging interaction that may cover conceptual issues and control flow, historical notes, good usage examples, language concepts and syntax and the concepts, use and syntax of a dozen ancillary tools. By examining system code, experts reconstruct and describe the concepts within, the distributed control flow the design rationale, and point out examples of good usage, bugs, and historical quirks. In their explanations, experts emphasize the information that is unwritten or
hard-to-find – what are useful information resources, what is the design philosophy, when and how to use complex tools, how to express a concept, or how to interpret an obscure code segment.

We believe that the interaction’s value to the apprentices comes from (1) the situated nature of the experts’ explanations and suggestions, (2) the range of topics covered and tools demonstrated while answering an initial question,(3) the experts’ emphasis on hard-to-find information, and (4) the collaborative conversational style that allows the learning to be tailored to the apprentice.

What about experts? The study proposes a view of programming expertise that ascribes their effectiveness not only to their knowledge of generic programming concepts and strategies, but to detailed knowledge of how these are expressed in their programming languages; the principles, design rationale, history and the quirks of the software framework they work within; proficiency in the use of broad range of ancillary tools; and knowledge of and appropriate use of human and written information resources.

This more encompassing view of programming expertise may help explain the manyfold individual differences in programming productivity. It suggests strategies to reduce the effort required to retool oneself from one environment to another, and as a side effect, the barriers to developing expert performance.

Studies need to evaluate the comparative benefits of interventions such as a crash course for programmers in the new environment, a more traditional apprenticeship model, or simply an improved programming environment. One can imagine a crash course in expressing oneself in the language and using the productivity tools, a trouble-shooting guide, and a crib-sheet for the couple dozen standard tasks. On the other hand, perhaps there should be a more traditional apprenticeship model, with apprentices working alongside the experts, getting immediate help with obstacles and observing the experts strategies, and tool usage. Or one can try to fix the problems, to rationalize the programming toolset, and transmitting the currently unwritten information about the tools, language, system and information sources.

Future work should also focus on evaluating the trade-offs between self-directed learning and more formal tutoring in such an environment, and teach apprentices how to make the best use of experts. Given the crucial mentoring role of the experts, we need to recognize and value that role and explicitly teach the social, pedagogical and cognitive skills required them to mentor their knowledge and their metacognitive strategies.

7 Acknowledgements

Many thanks to Robin Jeffries for supporting this research and for her valuable comments on drafts of this paper. I am also very grateful to the anonymous study participants who gave their time to this study, who took care to turn on the tape recorders, and who were willing to share their work with us.
8 References


