MANAGING COMPLEXITY BY SUPPORTING KNOWLEDGE GROWTH IN DESIGN AND DEVELOPMENT PROJECTS

Henrik Gedenryd

Lund University Cognitive Science
Kungshuset, Lundagård
S–223 50 Lund, Sweden
E-mail: Henrik.Gedenryd@fil.lu.se

A cognitive design theory is introduced, together with a set of tools for software development, based on the theoretical principles. The theory springs out of an analysis of the role of domain-specific knowledge, and how it develops in design and development projects that deal with complex target domains. It is shown that expert performance depends heavily on elaborate domain-specific knowledge. Two metaphors are introduced to explain how individuals develop their knowledge and skills: "from novice to expert," and "the person as an informal scientist." Because of the inherent complexity in large projects, participants' domain-specific knowledge evolves throughout their entire duration. The design process should adapt to the conditions of knowledge growth, since the project's success depends highly on it. Such a condition is that knowledge is immature and incomplete for quite some time. Furthermore, the design process should continuously support the growth of knowledge. The set of tools described addresses these issues in software development projects. It uses demonstrational programming methods, as well as programming-with-examples. The introduced theory affirms the cognitive value of these techniques, and also serves as a framework for understanding iterative design.

INTRODUCTION

The arrival of human–computer interaction (HCI) has made many end-user computer applications much more usable than they were before. This success has been possible because knowledge has been brought into application design from areas beyond computer science itself. However, there is an area of computing where this has not happened yet—that of programming and software development. The major problem in software development is still after thirty years that as programs grow, they become exponentially harder to make: Consider the task of producing a complex software system, say one that runs digital telephone networks, with hundreds of switching stations, and so forth. This necessarily involves very many sub-parts with intricate relations among them, and many different conditions and scenarios that must be handled well. Once the software grows, the difficulty of making it work grows not linearly, but exponentially. This is known as the 'complexity barrier.' A consequence is that while the computer industry has been pumping out new machinery at an overwhelming pace, progress in software has been very hard-earned. In other words, we are still waiting for exciting advances in software development methodology. Our belief is that bringing in knowledge on the cognitive aspects of this issue might work for software development, as it has done for 'traditional' HCI.

Computer science itself has of course approached the complexity barrier. The prototypical scheme is that of Software Engineering: trying to free the task from creative and artistic aspects, thereby reducing it to an automatic engineering process. This attempt at total mechanization has proven unsuccessful. Partly as a reaction to this approach, the human values in software development have been pointed out. Contrary to the engineering perspective, it explicitly acknowledges that the process involves creativity and artistry, and thereby affirms the individual as the fundamental asset in all design and development projects. The computer scientist best known for having defended the 'philanthropic' point of view, and who has also addressed the complexity barrier from this perspective, is Naur (1988). Further, work on the psychology of programming can be regarded as the ancestor of HCI as we know it (Curtis, 1985, Weinberg, 1971). Some early pieces of more HCI-oriented work were Smalltalk (Goldberg and Robson, 1989), Logo (diSessa, 1986a) and Pygmalion (Smith, 1977, 1993). The current Boxer project at Berkeley aims at creating a programming language based on principles of human cognition, and can be regarded as a successor of the work on Logo (diSessa, 1986a, 1991). 'Demonstrational programming' is an HCI

Lund University Cognitive Studies – LUCS 23 1993. ISSN 1101–8453.

topic-of-the-day, with a recent compilation of all relevant work in the field in (Cypher, 1993). However, most such projects do not address general-purpose programming.

Design activity as a cognitive process

The project described here concentrates on cognitive aspects of the development process. We believe that bringing cognitive science into shaping development processes and tools, part of the complexity barrier can be torn down. That is, we take human cognitive function as the outset for our effort to improve on design methods and tools. We want to point out the role of *knowledge* in managing complexity. There are many other important factors, like social patterns and communication within design groups, and so forth, but they are beyond the scope of this project. Because of this, and for the sake of simplicity, only the cognition of individual people is considered. To keep this in mind, we will refer to the person doing design and development as *the individual* from here on.

Anyone who writes programs will agree that their personal skills cannot be replaced by sophisticated tools or engineering methods. The same goes for designers and other similar professions. Any of these can therefore rightly claim that it is their *knowledge* and skills that are the most valuable assets in their projects. These intuitions justify our attempts to improve on design by focussing on its cognitive aspects, and knowledge in particular. There are, however, theoretical arguments as well. Initially, we will only briefly consider some of them.

Conceptual Models. First, we have theory on conceptual models; the well-known claim within HCI that the most important condition for someone to use a tool effectively is to have a good understanding of how it works (Norman, 1986, 1988). All of this is of course related to research on mental models (Gentner and Stevens, 1983, Johnson-Laird and Byrne, 1991). In the present context, this translates into that to do good design and development, the most important aspect is to have thorough knowledge (i.e. “a good conceptual model”) of the subject of work and its context, or what we will refer to as the *target domain*. This includes not just ‘shallowly’ knowing a great deal of facts, but to ‘deeply’ comprehend the subject. It cannot be stressed too much that having a good conceptual model means to really *understand*.

Domain-Specific Knowledge. Second, we have novice–expert studies, where it is well accepted that differences in knowledge cause the performance differences that have been studied. “In short, problem solving in a domain depends heavily on the quality and quantity of the problem solver’s *domain-specific knowledge*.” (Mayer, 1992, p. 413). Here we should understand ‘problem solving’ in a liberal way, including open-ended, ‘creative’ activities like design. The ‘domain’ includes not only the software product, but also its *context*. That is, everything that is related to the problem and relevant to it. This includes a great many things. In our telephone network example, this means knowing how such a software system operates, what it should do, what problems it addresses, the setting in which it will be used, the difference between various such settings, and so forth.

Programming as Theory Building. Third and last, the role of knowledge in software development projects has been explicitly considered in an article by Naur. There, among other things, he recounts a large project where the produced material of the original team, i.e. annotated program texts, documentation, and “much additional written design discussion,” was handed on to a new team that would make extensions to the initial product. The material supplied turned out to be inadequate for the new team to be able to do a good job. Naur concludes that “at least with certain kinds of large programs, … [successful work] is essentially dependent on *a certain kind of knowledge* possessed by [the project members]” (Naur, 1985). To stress these aspects, he introduces the metaphor of programming as theory building. Talking about theory stresses the point made in the previous paragraph, i.e. that domain knowledge goes far beyond just the software itself.

Having both intuitive and theoretical support for focussing on cognitive activity in programming and design, how do we best characterize it? A design process extends over a long period of time, often several months. So, we should look at what happens over time with respect to cognition. The best way to characterize this is by saying that the design process is a *learning process*. This is not at all a new idea; see e.g. (Bannon and Bødker, 1991). Learning is extra important in projects that deal with highly *complex* target domains, because a great deal of time will be spent on learning and understanding all the aspects of knowledge we enumerated above. Thereby, much work will be done while the individual still has restricted and immature domain-specific knowledge, and this must be taken into explicit account in the work process. A central aim of the project described here is to address the process of acquiring this domain-specific knowledge, from the perspective of cognitive science.

Knowledge can be seen as an ability and a potential to act in certain ways, and learning as how this potential changes. This is a highly abstract view of the long-term aspects of the design process. The complementing, concrete and short-term activity is the situated work as it really happens, and the corresponding cognitive processes. We may call this the *act* of problem solving. This is when the

individual is working on the telecommunications project. These two sides of cognition cannot be totally separated from one another. It is through the actual work that relevant knowledge becomes observable, since the knowledge in itself cannot be observed, and much of it is even unavailable to self-report and introspection. Therefore, we cannot consider learning without also saying something about situated design activity "as it really happens." A theory of design knowledge must therefore consider how such knowledge is put to work, for instance by showing how it materializes in the designer's actions. Still, that is beyond the scope of this presentation.

Knowledge development

Concentrating on how knowledge develops in programming projects and design activity in general, there are some connections with other areas of research. As we present these here, the nature of our task will become clearer. The phrases 'software development' and 'design activity' have already been used somewhat inconsistently, because the cognitive processes of knowledge development are essentially the same in both. There are several occupations that belong to this category: authors, film producers, architects, engineers, and so forth, but perhaps the most interesting kind of work is *scientific research*. There, knowledge corresponds to theories, and hence we may draw upon work in the philosophy of science and knowledge, which is a very valuable source. There, a great deal of work has been done on what theories are, how they develop, and what their place is in everyday scientific work. One very useful piece of material that is known far outside the domain of philosophy is Kuhn's paradigm theory (Kuhn, 1972). He even explicitly discusses cognitive aspects of scientific theories, see e.g. pp. 44–45. A related and very interesting issue is that of explaining human performance by using a 'human as informal scientist' metaphor (Nisbett and Ross, 1980). This metaphor outlines the activity of knowledge development in this way: we make observations, collect them, draw conclusions of general patterns from them, and establish cause-and-effect relations. Thus, to understand how knowledge develops we can expect to draw highly upon studies of the process of doing science. Additionally, as previously mentioned, the article by Naur (1985) is foresightful on the parallels between scientific activity and software development. Also note that the project described here itself involves knowledge development, since it is an example of developing a scientific theory.

A good metaphor besides the 'everyday scientist' view of humans, is the 'novice to expert' metaphor of knowledge development. It is fruitful to think of the individual as, at the outset of a project, a novice *in the project's target domain*, developing into an expert on the domain as work proceeds. But does not the individual often already have extensive experience in the target domain? Yes, that is true—but doing something you are well familiar with is not inspiring. Once having built a telephone switching system, you do not want to rewrite it from scratch, but reuse as much of it as possible. People with the kinds of job mentioned above want to "move on," and develop their skills and knowledge all the time. Ideally, they should use computers as tools to automate routine chores, freeing time for more stimulating tasks. Such tools should for example empower them by allowing them to easily generalize an existing system to suit a new set of conditions. Reuse should not depend critically on planning it in advance—how do you know if and how you will recycle your work in the future? People working in this way would thereby always be both novices and experts at the same time: novices on the new conditions, but experts on their previous work. The terms 'novice' and 'expert' will be used metaphorically to refer to the individual's status in terms of knowledge development—hence, being expert does not mean having ten years of experience; it only refers to the nature of the knowledge possessed, which we will now describe.

Expert Knowledge. Expert knowledge is highly complex and well integrated, both within itself and with related material. It is also embraces a great number of aspects: There are relevant concepts, a certain terminology, prototypical examples of problems and applications that have accumulated through experience, effective working methods and strategies, and so on. Much of this knowledge is tacit. Further, *understanding* means that the relevant knowledge is extensively organized and structured, so that experts know what is central and important to the subject, and what is auxiliary. (Again, we emphasize the value of deep understanding as opposed to just shallowly remembering a large number of facts.) This has been demonstrated in problem solving in physics (Chi, et al., 1981), and in many similar studies. Novices classified problems by surface features, such as 'inclined planes.' Experts instead attended to underlying principles, such as 'conservation of energy,' or 'Newton's Second Law.' These principles convey the right solution strategy, thereby enabling the experts to move directly towards an answer. It also definitely qualifies as a good 'conceptual model' in the previously discussed sense. The idea of mental models was originally proposed as the mechanism that enables humans to understand, predict and reason about the world they live in (Craik, 1943), or what we today would call a general theory of human cognition. Nowadays, the term is mostly used in more restricted senses, such as how we conceive of calculators, or reason by deduction and inference (Gentner and Stevens, 1983, Johnson-Laird and Byrne, 1991). So expert knowledge is complex

enough to be well characterized as a *personal theory*, and experts' mental models are so sophisticated and elaborate that they should be characterized as 'systems' of the complexity addressed by systems theory, e.g., (Weinberg, 1975). As mentioned, important aspects of expert knowledge are integration and structure, since they correspond to experts' superior 'deep' comprehension.

Novice Knowledge. The individual's knowledge at the outset of a project can be described as 'novice knowledge.' It is characterized by the opposites of the aspects of expert knowledge. Hence, it is 'shallow,' consisting of loose bits of information that are not well integrated, neither between themselves, nor with related knowledge. On the contrary, it is typical of the novice not even to know what other knowledge *is* related and relevant. The bits that exist are the few notions that the individual has yet come upon. These are too few to allow any kind of systematic organization. Therefore, it is also impossible to know what pieces of information are most relevant and characteristic, in order to extract the most important aspects of a problem. Accordingly, novices cannot systematically explore their target domains. They cannot even formulate the relevant questions, or state their goals more than loosely. Thus, a telecommunications novice cannot formulate a problem statement for a switching application—stating a goal adequately requires knowing what is important and relevant for such a switcher, and what is not. By necessity, a novice's problem statement would be superficial and concentrated on the shallow aspects of the network system. Remember how novices characterized physics problems by surface features, while experts concentrated on relevant aspects. (Novices' understanding corresponds to what diSessa calls 'distributed models' (1986b, 1991).)

From Novice to Expert. In order to do a successful job, our individual has to acquire good domain knowledge, distinguished chiefly by great integration and elaborate structure. To do this, it is necessary to find pieces of information, tie them to each other, and elaborate the developing structure by carving out details. If a single most important part of knowledge development should be selected, a preliminary review of theory suggests *building good, well integrated structure*. In this way the individual will be able to tell what is important and what is not, among other things. Until this thorough grasp of the problem is established, there is no way of knowing whether progress is being made, etc. The primacy of mature knowledge structure has also been established empirically (Fix, et al., 1993).

Supporting Knowledge Growth

In order to improve the processes of design and development to suit knowledge development, we need new models for these processes. These models should adapt to, support, and ideally even augment knowledge development. By adopting the view that design and development is a kind of theory building, we get a natural connection between knowledge development and contemporary HCI-work. The original idea about mental models was that people use them as theories by which they understand their world (Craik, 1943). Seen this way, knowledge growth in design and development is an example of personal theory building (Nisbett and Ross, 1980): In the beginning, our individual observes some telephone network systems, collects memories in the form of examples and so forth. As work progresses, the task becomes to establish patterns out of these examples, such as general properties and behaviors of telephone networks. This general knowledge also makes up the structure and organization of knowledge. General patterns reveal what network aspects are very common, and therefore highly relevant. Rare details are less important. Thus, abstract patterns in knowledge make up the structure and organization of the domain knowledge, since they establish relevance and so forth. Ultimately, the individual tries to establish cause and effect relations that let events be predicted: Dialling a number causes a line to be hooked up, which in turn means that proper charges must be calculated, and so on.

The principal truism here is that we cannot know anything before we know it. How ever banal this principle may sound, we often fail to follow its consequences. A typical mistake is to try to make a complete, detailed problem statement or goal specification prematurely in a project. That will not be possible until quite late, since making expert formulations requires having expert knowledge. Another example is that far too many existing software development environments violate this principle by requiring "a complete system." Such environments require considerable effort to be spent on achieving basic or even rudimentary functionality, so that when the work can finally start to be tested and evaluated, the important insights that are made are long overdue. Often quite severe flaws are discovered. Imagine that a basic switching system must be finished before work on special communication services can be tested and evaluated. An oversight discovered then may require a major redesign of the basic system that has already been completed. This will require that piles of produced work have to be thrown out and redone. An almost proverbial observation states what is well known to everyone that has written a program beyond the most trivial kind (Brooks Jr., 1975, p. 116):

Plan to throw one [version] away; you will, anyhow.

This expresses the need to do practical work on your ideas, in order to learn about your task, to test your ideas, and to find out what you do not know. It is also the most drastic argument we have for our claim that knowledge is the important factor in all design and development projects, formulated by a genuine authority on the matter. However, it is not the final saying on the issue—it merely expresses an adaptation to the tools available. In concentrating on knowledge growth, it is precisely our task to address this problem. Good tools must facilitate learning and understanding, and do this by allowing the individual to work on the task in a manner that matches the prevailing level of knowledge. Early on, the informal scientist is still conducting observations and collecting example data. At this stage when knowledge is scant, it must be possible to work on be example cases, instead of on general and complete rules and principles—anything else is premature and a waste of time. Further, there must be no requirements for the pieces to fit together. We may call it support for pre-structured work. This is what would allow work on our hypothetical telephone network system to begin with trying out the various services before deciding on the core system's overall design principles. By necessity, there must be an interaction between on the one hand working on small, concrete parts, and on the other hand developing the fundamental and general principles.

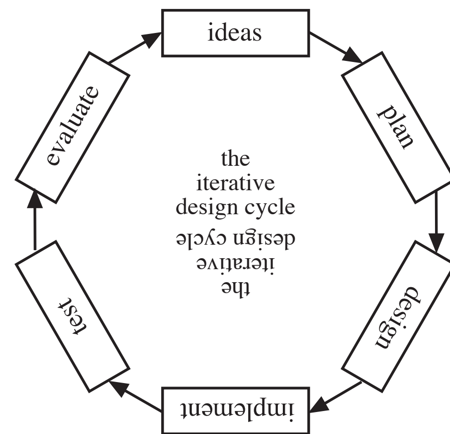
Good understanding depends fundamentally on how material is represented 'externally': for example in network maps, figures of system structure, functionality check lists, visualization of system functions and graphs of network traffic, etc. (see Cox (1993) for an extended discussion). The role of external representations for cognition has been established in many fields, for example in deductive reasoning (content-specific reasoning), in learning and education, and in situated action ("knowledge in the world") (Johnson-Laird and Byrne, 1991, Larkin and Simon, 1987, Mayer, 1992, Norman, 1988). An important example is, of course, direct manipulation interfaces. All phases of development depend heavily on good representation. Tools should support great flexibility in choosing visual representations and manipulation techniques, since it is essential that external representations match how the individual thinks about them, thereby reducing *semantic distance* (Norman, 1988) At the very least, it should be as easy to have graphical representation with direct manipulation, as having textual representation with keyboard input. This would allow our individuals to work on computer representations that resemble how they conceptualize them, say connections displayed on maps instead of in tables with the computer's representations (internal names, reference numbers, etc.).

While it is important to work on the various pieces of knowledge you have, sooner or later you have to start fitting these pieces together. Without organization there is no theory, only a bundle of ideas. As discussed above, building a coherent structure is the most valuable aspect of all for knowledge to develop. Therefore, while working on the bits that are known, it is essential to frequently take a step back so as to maintain and develop a notion of the overall structure. When ideas are starting to join together for the individual, the task becomes to build this organized structure that is to the personal theory what the framework is to a house. This phase is a mighty leap on the road to becoming an expert, and corresponds to the phase of establishing a *paradigm* in Kuhn's (1970) sense: Once it is in effect, the rest is just to fill in the missing details into the framework, which is fairly straightforward to do, since the overall structure has been put together. A good set of tools supports both building the framework and filling in out the details. The former would be something similar to mind-mapping tools, or more generally, sketch-pads. Since mind-maps, sketches, and personal notes in general are highly individual and they are very important external representations, tools for this purpose must be highly flexible. Firstly, everyone has their own personal style in these matters, and secondly, no two projects have similar requirements: making a telephone network system is different from making an accounts-receivable package. The idiosyncrasies of both people and projects are the reason why fourth-generation and CASE tools are often felt to be too restricted to be practically useful, since they lack in flexibility. It is important to remember that pen and paper are superior in respect to flexibility, and are unlikely to be superseded by computers as the designer's primary thinking tool. However, since the design material is produced on computers in the cases we are considering here, it becomes impractical to keep them organized by using paper and pen. This is a respect that could give computerized organization tools an advantage over the former: For it to happen, the collected examples, tests, and data should be equal members in the emerging structures that the individual sketches on. Testing procedures, network capacity estimation formulas, and everything else produced on computer would integrate seamlessly with documentation, design discussion and sketches. Then, such tools could become an important complement to the notebook. This goes as much for any kind of computer-assisted development such as mechanical, architectural and graphical design, authoring books (and computer media, of course) and so forth, as it does for software development.

Finally, there is the last phase of projects, when theory has been established and most issues seem clear and simple, although work is not yet finished. This is similar to what Kuhn calls 'puzzle-solving,'

scientific research within the normal paradigm: no revolutionary breakthroughs come out of it, but it still requires respectable skill. On this stage, work often becomes routine and boring, so here tools may help by enabling experts to automate repetitious and predictable tasks in the manner discussed above. Thus, they may produce meta-tools that are not part of the finished product, but yet of great value—and even necessary in all larger projects. Such tools could be used for automatically generating phone-calls during tests, automatic data consistency and error checks, statistics gathering for net traffic, and so forth. Also, working on meta-tools is a good way of analyzing one's data and constructing suitable, general patterns.

As discussed, important insights may cause a lot of produced material to become useless. This is especially true late in the process, when knowledge becomes very good, and the really good ideas and deep insights start to arrive. This is when *iterations* occur, in terms of modern design theory: Iterating means going backwards, redoing work to accommodate to *new* knowledge that one has acquired: 'This implementation is no good, redo it." etc. (See figure 1.) During implementation or test, etc., realizing that *more* knowledge must be generated is another kind of iteration—requiring work to 'go back' to earlier stages in the iterative sequence: "First we'll have to figure out how to handle this unexpected kind of overload." In either case, effort has been wasted, and therefore iterations should be minimized. This is why all parts of the project ideally should run in *parallel*, being at similar levels of progress to as high an extent as possible. This reduces the risk of encountering problems in parts that are lagging behind, and that will cause long proceeded work in other areas to be discarded. This does not only mean parallel work on different modules of the product, but also on different activities such as planning, implementation, testing and evaluation. If any of these lag behind the others, a problem encountered there may cause major iterations to become necessary. Likewise, one line of work running ahead of the others may have to be discarded as the others advance. Instead, the ideal way to work is by letting the different sub-tasks interact: implementing this generates a new idea, testing that yields an important design hint, and so on.



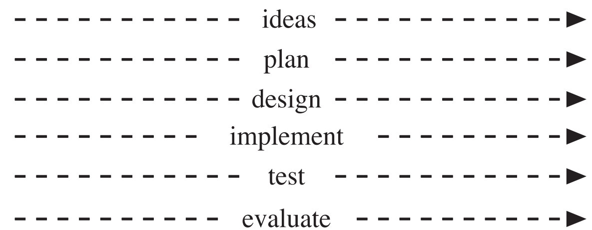


Figure 1: parallel and iterative development

We may thus summarize: Work on the product should not run ahead in any part, but all aspects should be brought forward in parallel, thereby allowing them to interact. Theory and understanding, however, must continuously strive forward and preferably lie substantially ahead of practical work, since knowledge is such a fundamental prerequisite for successful progress on the product itself. However, practical work is very important for gaining necessary insight and should be conducted to interact with the advance of theory. Therefore, when practical work and hands-on experience produces understanding, the material output is wasted—but the effort is not, since it yielded knowledge and insight. Time may only be wasted when it is spent on producing material and does not develop understanding—unless the understanding later turns out to be irrelevant, but that problem has already been discussed.

In any case, the basic lesson learned is that the role of domain-specific knowledge and knowledge development should be explicitly considered in design projects that are large and/or deal with a complex problem domain, and we have suggested some important techniques, such as parallel progress. Because of the complex nature of the projects we address, we do not believe that any reasonable empirical studies can strengthen our theoretical claims.

An important part of the project described here concerns exploiting the ideas in experimental implementations of programming tools. There is

only room here for a brief outline of the main directions of work, which includes a set of tools and a 'metaphysics,' or representational model, that materializes as an 'abstract machine.'

As already suggested, there is a stress on supporting 'pre-structured' work. Inspirations are exploratory programming systems like Smalltalk and Lisp, and rapid prototyping tools. Both allow playing around with ideas, too often overlooked as an important catalyst for discovery. (Do you think that children's play is just for fun and without impact on their development?) Our approach is a set of high-level tools for knowledge discovery, or what we might call "the informal scientist's laboratory." These tools are based on direct manipulation and demonstrational techniques, thereby supporting directness, concreteness and interactivity; important factors for good problem solving (Mayer, 1992). Spreadsheets are perhaps the most successful examples of what such tools are like (Kay, 1984). Also, Shneiderman's survey of "Information Exploration Tools" gives a good idea of what such a set would contain (Shneiderman, 1992, ch. 11). The challenge lies in finding a small set of tools and techniques that are sufficiently general, and further that these may be brought into higher levels of sophistication as work progresses. This way of working is well suited for techniques of 'programming with examples' (Cypher, 1993, Smith, 1977). Early on, observation and data collection is complemented by working out examples for what should happen. Later, as knowledge is elaborated, these examples can be extended and worked out in greater detail, thereby becoming 'real programs.'

The 'metaphysics' is an important part of our approach, that materializes as an 'abstract machine'—a paradigm for computer processing. A computational metaphysics must contain models for representation, processing, generalization and abstraction: Lisp represents by symbols and lists, processes in a functional style, and generalizes by symbolic variables; Pascal has data structures (records), sequential algorithms and mathematics-style variables; and Prolog has basically the same representations as Lisp, but instead processes with production rules, and generalizes by unification—and so forth. Our strategy is to build a programming metaphysics that suits humans' cognitive processing as much as possible, thereby striving to reduce the semantic and articulatory distances of programming (Norman, 1986, 1988). Using the ideas about humans as informal scientists as our starting point, we adopt Craik's general argument that mental models comprise our informal theories, and substantiate it with techniques from systems theory (Craik, 1943, Nisbett and Ross, 1980, Weinberg, 1975). We believe that the techniques of systems theory are invaluable to anyone that conducts systematic studies and development of more complex nature. Their universal character matches our concerns about reducing the need for mastering idiosyncratic programming techniques (see figure 2). If our ideas become successful, knowing how to program becomes having a good command of how to build models systematically—general-purpose knowledge valuable to most professionals—instead of a mandatory undergraduate course far removed from the needs of those who study industrial or graphic design, chemistry, architecture, human factors, and so forth…

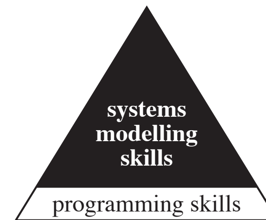


Figure 2: general-purpose vs. domain-specific skills

Summary

This description of knowledge development, with our theory and the accompanying implementations, has focussed on explaining the theoretical parts, since these may be of direct value to designers and developers. This has necessarily been on the expense of thoroughness on the implementational side. Anyway: The starting point of the research described here was to consider design and development as a *cognitive* process, leading us to concentrate on aspects of *knowledge*. We have shown that *domain-specific knowledge* is a primary determinant of expert performance in problem solving and design. In projects dealing with highly complex domains, as HCI and other software development projects usually are, domain knowledge is so complex that it takes participants a very long time to master it. Taking these two observations together, it becomes clear that the design process should be adapted to the conditions of knowledge growth. We have presented a theory of knowledge development, together with a model of the design process that is based on this theory. Also, our accompanying experimental implementations have been briefly considered. The latter address in particular the knowledge development issues in software design and development. We hope to have shown the role of knowledge and the value of addressing cognitive aspects of design and development processes.

References

Bannon, L. J. & Bødker, S., (1991), "Beyond the interface: encountering artifacts in use". In J. M. Carroll (ed.) *Designing Interaction: Psychology at the Human-Computer Interface*, Cambridge University Press, Cambridge, UK.

Brooks Jr., F. P., (1975), *The Mythical Man-Month: Essays on Software Engineering*, Addison–Wesley, Reading, MA.

Chi, M. T. H., Feltovich, P. J. & Glaser, R., (1981), "Categorization and representation of physics problems by experts and novices", *Cognitive Science*, **5**, 121–152.

Cox, R. & Brna, P., (1993), "Reasoning with external representations: Supporting the stages of selection, construction and use". In *World Conference on Artificial Intelligence in Education 1993*, Edinburgh, Scotland.

Craik, K., (1943), *The Nature of Explanation*, Cambridge UP, Cambridge, UK.

Curtis, B. (ed.), (1985) *Human Factors in Software Development*, IEEE Computer Society Press, Washington, DC.

Cypher, A. (ed.), (1993) *Watch What I Do: Programming By Demonstration*, MIT Press, Cambridge, MA.

diSessa, A. A., (1986a), "Models of computation". In D. A. Norman and S. W. Draper (ed.) *User Centered System Design: New Perspectives on Human–Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ.

diSessa, A. A., (1986b), "Notes on the future of programming: Breaking the utility barrier". In D. A. Norman and S. W. Draper (ed.) *User Centered System Design: New Perspectives on Human–Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ.

diSessa, A. A., (1991), "Local sciences: Viewing the design of human–computer systems as cognitive science". In J. M. Carroll (ed.) *Designing Interaction: Psychology at the Human–Computer Interface*, Cambridge University Press, Cambridge, UK.

Fix, V., Wiedenbeck, S. & Scholtz, J., (1993), "Mental representations of programs by novices and experts", *Proceedings of ACM INTERCHI'93 Conference on Human Factors in Computing Systems*, 74–79.

Gentner, D. & Stevens, A. L. (ed.), (1983) *Mental Models*, Erlbaum Associates, Hillsdale, NJ.

Goldberg, A. & Robson, D., (1989), *Smalltalk–80: The Language*, Addison–Wesley, Reading, MA.

Johnson-Laird, P. N. & Byrne, R. M. J., (1991), *Deduction*, Lawrence Erlbaum Assoc., Hillsdale, NJ.

Kay, A., (1984), "Computer software", *Scientific American*, (March):52–59.

Kuhn, T. S., (1972), *The Structure of Scientific Revolutions*, Unversity of Chicago Press, Chicago, IL.

Larkin, J. H. & Simon, H. A., (1987), "Why a diagram is (sometimes) worth ten thousand words", *Cognitive Science*, **11**, 65–100.

Mayer, R. E., (1992), *Thinking, Problem Solving, Cognition*, Freeman, New York, NY.

Naur, P., (1985), "Programming as theory building", *Microprocessing and Microprogramming*, **15**, 253–261. Also in (Naur, 1988).

Naur, P., (1988), *Computing: A Human Activity*, ACM Press, New York, NY.

Nisbett, R. & Ross, L., (1980), *Human Inference Strategies and Shortcomings of Social Judgment*, Prentice–Hall, Englewood Cliffs, NJ.

Norman, D. A., (1986), "Cognitive engineering". In D. A. Norman and S. W. Draper (ed.) *User Centered System Design: New Perspectives on Human–Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ.

Norman, D. A., (1988), *Design of Everyday Things*, Basic Books, New York, NY.

Shneiderman, B., (1992), *Designing the User Interface: Strategies for Effective Human–Computer Interaction*, Addison–Wesley Publishing Co., Reading, MA.

Smith, D. C., (1977), *Pygmalion: A Computer Program to Model and Stimulate Creative Thought*, Birkhauser Verlag, Basel, Switzerland.

Smith, D. C., (1993), "Pygmalion: an executable electronic blackboard". In A. Cypher (ed.) *Watch What I Do: Programming by Demonstration*, MIT Press, Cambridge, MA.

Weinberg, G. M., (1971), *The Psychology of Computer Programming*, Van Nostrand Reinhold, New York, NY.

Weinberg, G. M., (1975), *An Introduction to General Systems Thinking*, Wiley & Sons, New York, NY.