Methods in user oriented design of programming languages

Clayton Lewis
Department of Computer Science
University of Colorado, Boulder
Fellow, Hanse-Wissenschaftskolleg
clayton.lewis@colorado.edu

Abstract

Card and Newell, in an influential 1985 paper, described programming languages as "obviously symmetrical" between programmer and computer, and called for balanced investment on the programmer and computer sides of the design space. But the design space is in fact more complex than that, with important impacts of purpose, as well as of programmer and computer, on effectiveness. Further, each part of the space is fragmented into many distinct areas, reflecting consequential differences in people, purposes and computational setting. Empirical methods face challenges in spaces of this complexity. As suggested by research on the role of mechanisms in scientific thought, cognitive dimensions analysis is better suited to operate in this complex space than are empirical methods, and should be promoted and extended.

1. Introduction

> Millions for compilers but hardly a penny for understanding human programming language use. Now, programming languages are obviously symmetrical, the computer on one side, the programmer on the other. In an appropriate science of computer languages, one would expect that half the effort would be on the computer side, understanding how to translate the languages into executable form, and half on the human side, understanding how to design languages that are easy or productive to use. ... The human and computer parts of programming languages have developed in radical asymmetry.
>
> --Newell and Card, 1985

I've given talks for years built around the above quotation from Newell and Card, calling for greatly increased attention to user oriented design of programming languages (UODPL). But recently I've come to think that Newell and Card's framing of the issue needs substantial attention. The opportunity as they suggest it is too simple: all we have to do is pay more attention to the human side of the picture. The actual situation is much more complex, and, accordingly, the response needed from us isn't simple, either. Thinking through these matters, I will argue, provides strong support for the continued development of lines of work identified strongly with PPIG as an institution, and indeed for the defense of this intellectual approach against seemingly attractive alternatives.

The original impetus for this rethink came a few years ago, when I led an undergraduate seminar on UODPL. This would be easy, I thought, with the symmetry of Card and Newell's diagnosis as a starting point. As we reviewed and discussed the literature, however, I felt that work didn't fall into line in the way I expected, and that the picture became murkier, not clearer, as we progressed. Why?

The urge to do more with this question rose above threshold during the panel session, ‘PPIG in the wild - what should we be studying?’, at PPIG 2016. There two panelists, Meredydd Luff and Steven Clarke, presented sharply differing accounts of trying to do real UODPL work, on an academic project and in industrial development work respectively. Luff found no guidance in the PPIG literature, or elsewhere in the literature, because the seemingly relevant quantitative results were inconsistent, and thus incoherent. In particular, different measures of programmer effort were uncorrelated. Clarke, on the other hand, presented a gratifying story of real impact on the thought and work of developers, based on a combination of videos of actual programmer experience, and cognitive dimensions analysis. I'm suggesting here that it's the complex nature of the UODPL landscape that explains why the literature is so unsatisfying, and why Luff did not find what he was looking for, on the negative side, and (on the positive side) why Clarke has been so successful.

Based on this analysis I'll argue that some approaches to our subject will be more successful than others. In particular, those of cognitive dimensions style will be impactful, but generalizations based on quantitative data will be much less so, despite frequent calls for this particular kind of scientific work.

2. The complex landscape of programming language design

As suggested by the sketch of the language design landscape in Figure 1, Newell and Card identify two abutting regions, the programmer side and computer side. The computer side has been studied a lot, and the programmer side, not much. We need to get to work, and it's clear where.

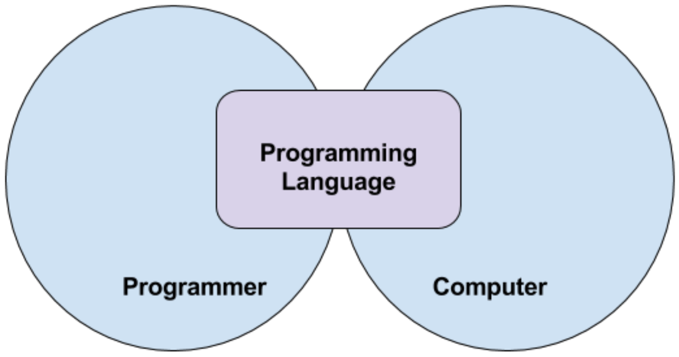

Figure 1: The design landscape in Newell and Card (1985)

Figure 2, below, shows the landscape as I now believe it to be. There are lots of differences.

One big change is the addition of a whole new region, that of "purposes". How well a programming language works is determined not just by how it is implemented, and who is using it, but crucially what it is being used for. This isn't news: we've thought since early days that FORTRAN is a better tool than COBOL for "scientific" work. But when we think about what "programming" is, we often lose sight of the differences. Perhaps talk of "general purpose" languages helps us forget. But surely we can accept that there aren't actually any "general purpose" languages, and never could be. Is Java "general purpose"? Would any sane person make a cat character dance and emit thought balloons in Java rather than in Scratch?

Another change is that all the regions are subdivided. The new "purposes" region is carved up into different kinds of target domains, of which I've suggested just a few in the figure. Many domains require mathematical reasoning of various kinds, including those calculations rooted in physics for which FORTRAN was designed. Others are mathematical, but not in the same way; FORTRAN isn't as well suited to logic or graph theory. A vast range of domains are hardly mathematical at all. The record structures that originated in commercial data processing in the assembler era, and gradually came into higher level languages, provide basic representational tools in a very wide range of important applications, but these are barely mathematical. Some target domains involve dynamics in essential ways, while many do not; concepts of timing, or of synchronization may or may not be needed, and be available. Some domains involve interaction, requiring another, different suite of concepts, and so on.

Someone wanting to understand what will go well or not so well for a Scratch programmer (to take just one example) can't overlook these variations in the landscape. Scratch provides quite direct support for scattered aspects of these domains, and little or none for others. The same is true for any other language one might name.

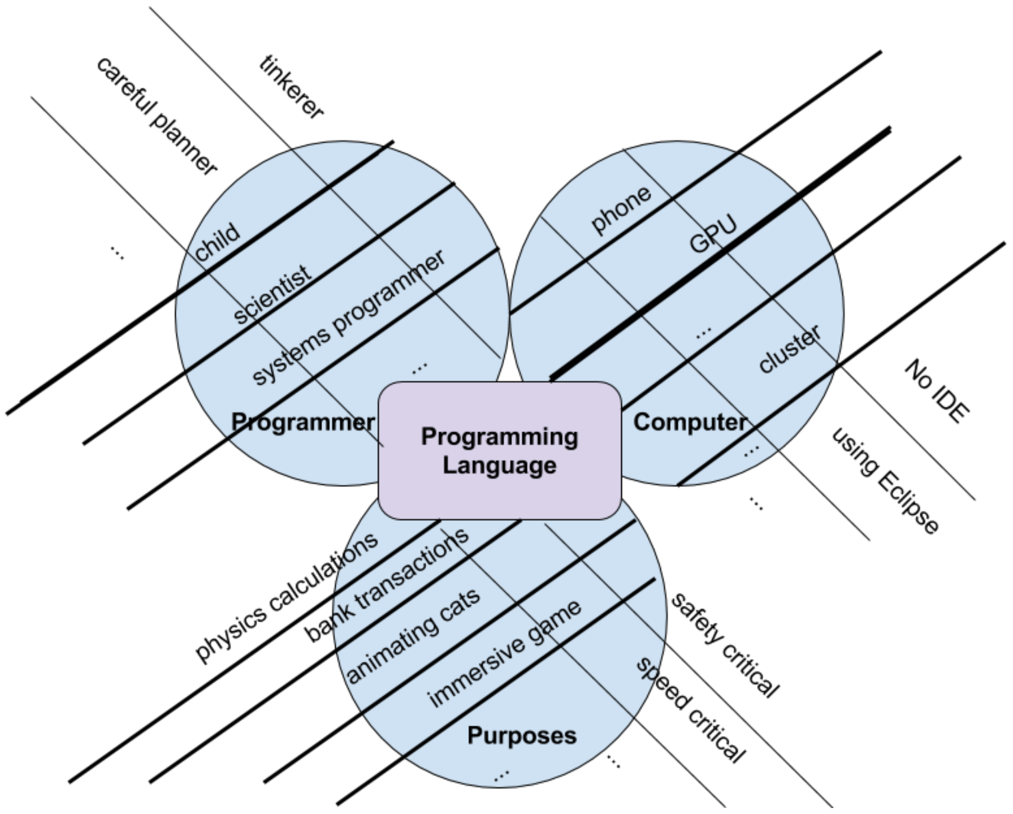

Figure 2: The larger, tessellated landscape

Cutting across these target domains is an additional set of contrasts, that might be called "contextual requirements". Someone creating an antilock braking system, or a defibrillator, must deliver an extremely strong correctness guarantee, while someone making a cat character dance need not. Thus in the former case it will be rational to require quite laborious conceptual work, if in return the needed strong correctness guarantee can be obtained; in the latter case it's silly and counterproductive.

Contextual requirements are themselves diverse. Some programs have to run very fast, others need not; some must interface with or exploit stipulated hardware (GPU, sensors, ...), or not; must interface with stipulated software; must be readily reconfigurable to work with new devices or programs; must be maintained so as to operate for many years, or not; the situation being addressed requires millions of lines of code, or just a few; and so on and on and on. Just as for correctness guarantees, any of these requirements will change the rational assessment of programming language features. Complications that can and should (must?) be avoided in some situations can't be avoided in others.

Not shown in the figure is another set of purpose distinctions. Are we observing someone thinking about a program, and trying to make a plan, or someone who has a plan and is writing code, or someone who has code and is trying to debug it, or someone who has someone else's code and is trying to debug that, or someone who has code and is trying to assess its correctness empirically, or someone who has code and is trying to prove its correctness, or someone who has code and is trying to extend it, or ..., or .... All of these distinctions are consequential, in that an aspect of a programming system can be good or bad for one kind of purpose, and bad, indifferent, or good for another.

The “programmer” region is crisscrossed with distinctions, too. Is the programmer a child? A systems programmer? A pensioner? A scientist? Do they like to learn by tinkering, or step by step? Do they like to understand what they are doing, conceptually, or would they be happier not to? Other differences, not shown, cut across these. Is the programmer working alone, or in a group? Are they knowledgeable about the mathematical aspects of the situation with which they are engaged, or not? Do they have years of experience with Python? It's obvious that any observations of programming by people drawn from this pool will be extraordinarily variable, and that deriving lessons from what is observed in one part of the pool about what might happen in another part will not be easy.

The computer side has its own subdivisions, only suggested in the figure. The challenges on phones are different from those on clusters. Further, we know, but often forget, that development environments and tools condition the effectiveness of language features. So does the language paradigm within which language features function, not shown. Imperative programming is not all of programming. Do we think that learner difficulties we see in imperative programming are a guide for what we would see in functional programming? Reactive functional programming? Logic programming? In object orientation, what is the impact of multiple inheritance? How about spreadsheet programming?

These complexities can be illustrated by a controversy of the past. In the 1970s two styles of electronic calculators were popular. Texas Instruments calculators used "algebraic notation", in which complex calculations were described using a notation similar to that used in written mathematical formulae. Hewlett Packard calculators used reverse Polish notation (RPN), in which arguments to an operation are entered first, followed by the operator. Which notation is better? Surely one must be better, and people argued about which that was.

The empirical results of user speed tests tended to favor RPN, or to show no difference (Card, 1979; Kasprzyk et al. 1979; Agate & Drury, 1980; Hoffman et al., 1994). The size of the difference depended on the details of the tasks, but never showed a clear advantage for algebraic notation. Yet RPN calculators have all but disappeared in the decades since these evaluations were done. What did they miss? The evaluations all used skilled users, often engineering students. For people generally, the familiarity of evaluating 2+2 that way, rather than as 2 2+, seems to carry the day. So, which notation is better? It depends on whether performance by skilled users is important, or familiarity in casual use.

The impact of the complexities in the present has been described by insightfully by Luff himself. In his paper, “Empirically Investigating Parallel Programming Paradigms: A Null Result” (Luff, 2009) he lists several obstacles to the empirical comparison of language features. Besides weak metrics for programmer effort, he cites the effects of learning, where test participants learn things from an earlier task that affects their performance on a later one; the effects of an old design being more familiar than a new one with which it is being compared; toy problem syndrome, the problem of test tasks being smaller and simpler than real ones; the variability among different possible test tasks; the difference between ease of writing and ease of maintaining code (that is, contrasting purposes); student test participants not being representative of real users; and others.

3. Methodological implications

3.1 Measurement considered hopeless.

If we consider a seemingly clear and simple notion, "programmer effort", against the tessellated landscape of purposes, people, and computational structures we can see that it is hopeless as a target for quantitative measurement. Thus Luff's frustration with available metrics is just what we expect. The question, "How much programmer effort would using feature F require,", or even "Would using feature F require more programmer effort than feature F'," are both in a rich class of questions in HCI, those that *have no answer* as posed, however clear, concrete, and practical they seem.

The type specimen of questions without answers in HCI is, "Tell me what colors I should use for these controls?" Are your users members of the public, about 5% of whom will have anomalous color vision, or military pilots who are required to have typical color vision? Do they include Chinese people, for whom red is the color of happiness, and white the color of death? Are there similar controls already deployed elsewhere in this system? How many and what controls need to be discriminated? and so on. As we'll see, and as many in the PPIG community have seen, there is a way to handle the design needs that lie behind questions like these, but it isn't a metric.

One's first instinct might be to deal with the tessellation problem by what we might call special purpose sampling. If programmer effort depends on many different factors, of audience, purpose, and computational setting, let's roll up our sleeves, determine what *levels* of all those factors are the ones we care about, and collect some measurements in just a small corner of the space. This is almost never workable, for multiple reasons.

First, measurement is a lot of work. Even within small categories (of people, for example) variability is often high, so a lot of data are needed for useful estimation. Only rarely will people judge that the impact of a small design feature warrants the measurement cost.

Second, even if cost is no object, the measurements will often be virtually worthless. The situations of practical interest will almost always take in different levels of our various factors. Perhaps we are interested only in 14 year olds, but will all the 14 year olds know nothing at all about programming? Will all their programs be interactive, or not? Will they be working individually, or will some work in pairs? Will some work at home?

Here we have to worry not only about these several levels, but about their *interaction*s. That is, we can't measure the impact of student age and knowledge, and then separately the impact of the setting in which they work. The impact of work setting may well be different for older students than for younger ones, and if measurement is our approach, we can't tell that without taking measurements for all the combinations. Given the complexity of the space of situations we are seeking to understand this is combinatorially hopeless.

All this is bad enough for us, but it is worse for a designer looking for guidance. Even if somehow we expend the resources to do a really thorough survey of the part of the landscape we care about, when a designer comes along and reads our report, will it shed any light at all on the problems they are working on? Very likely not. If they are interested in professional programmers, or university students, not 14 year olds, they're out of luck.

The situation is actually even darker than that. The designer wants to know the effort associated with a *new* language feature that they have just invented, one we could not possibly have collected data on. We have to accept that measurement can't tell us what we (and the designer) need to know.

3.2 The logic of cognitive dimensions

Why is Clarke happier than Luff? Because Clarke did not rely on measurement at all, but on a combination of cognitive dimensions analysis and observation. Let's focus first on cognitive dimensions.

Cognitive dimensions (see, among many papers, with many contributors, Green, 1989; Green and Blackwell, 1998) work by describing aspects of a situation that are likely associated with difficulties (and hence with errors, effort, and other costs.) For example, a *hidden dependency* is an aspect of a situation that will cause trouble when someone has to understand the consequences of some modification they might undertake, such as changing the declaration of a variable. A programmer might not see exactly what uses of that variable will be affected by the change; a programming environment might (or might not) provide information that makes the dependency more (or less) apparent.

Note that there is nothing in the definition of a cognitive dimension that is particular to audience, purpose, or computational setting. Hidden dependencies will affect children and scientists, high performing and low performing programs, and imperative or functional languages in the same way. It makes sense for a designer to examine their completely novel design, to see whether their users are likely to encounter issues of this kind. Even better, the logic of hidden dependencies includes within it guidance on remediating it. Thus a designer may learn not just *that* their design has a likely problem, but also *why* the problem occurs, and thus *how* it might be avoided.

Clarke's experience demonstrates that the sophisticated developers with whom he works find the logic of cognitive dimensions clear and compelling. They not only recognize the force of design critiques based on them, but also make such critiques themselves, and see the value of responding to them, resulting in improvements that are meaningful to them.

On the other hand, Clarke's presentation showed that his collaborators did not respond well when he gave them *data* showing low success rates by their users. But aren't data the gold standard? No.

In my view Clarke's coworkers were actually right to be resistant. As we've seen, the sample could hardly be representative, and worse, the data include no information useful for responding to the problems they suggest.

3.3 The role of observation

Clarke got started with videos that showed examples of people having the kind of difficulties that cognitive dimensions identify. Why was this important? I suggest there are two reasons. First, someone new to cognitive dimensions might well not see that the relatively abstract descriptions that Cognitive Dimensions provides can actually be applied in their situation. The videos show that they can, and can give an assurance that they are useful. Second (as this essay argues throughout) programming is an extremely complex activity. Even an experienced cognitive dimensions analyst can get things wrong, by not seeing issues that users actually encounter, or not seeing how users can in fact avoid an apparent pitfall. Observing actual user behavior provides a check on what the cognitive dimensions analysis suggests.

Does the need to make observations of users plunge us back into the combinatorial tar pit of different users, purposes, and so on? Don't we have to observe many different kinds of people, doing many different things? A little, but only a little. Let's take two cases. First, suppose we observe a few people, and there are no surprises. We can't be *sure* that there aren't problems lurking somewhere out there, for a user with some unforeseen misconception, or trying something we hadn't thought of. But we can have practical confidence that in the normal case things will be more or less OK. Why? Because we have formed a theory of what will happen in the normal case, and we don't see any reason to think that theory is wrong.

We have no ultimate assurance, of course; we never could. Measurement wouldn't provide it, either.

Second, suppose we make some observations, and there is a surprise. People have trouble that we didn't expect, or perhaps a trouble that we anticipated doesn't materialize. Either way, here we have learned that our theory is wrong. If the issue is consequential, we need to fix the theory, or modify the system to steer around the issue. Note that in neither case will it help us to blanket the space with sampled users and tasks, in the teeth of the combinatorics. We might observe a few more people, hoping to see more examples that can help us to understand what's happening, but surveying that huge space won't help with that.

3.4 The situation in science generally

One view of science is that it is, or ideally should be, a parade of randomized controlled trials. For some, only such experiments produce reliable knowledge; advocates of "evidence based" approaches in medicine and education promote this idea (see for example National Center for Education Evaluation and Regional Assistance, 2003). But studies of actual scientific practice give a picture more compatible with cognitive dimensions logic.

Darden and colleagues (Machamer, Darden, & Craver, 2000) have documented the pervasiveness of *mechanisms* in scientific work and thought, especially in biology. Mechanisms explain phenomena of interest by describing the entities, relationships, and activities that fill out a narrative of how a phenomenon is produced. Darden (2002) also identifies *mechanism schemata*: "A mechanism schema is a truncated abstract description of a mechanism that can be filled with more specific descriptions of component entities and activities" (p. S356). Darden describes how these schemata provide a structure for scientists seeking to understand a new phenomenon. "Once a schema is chosen or sketched in a discovery episode, then the task is to find the entities and activities, or modular groups of them, that play the roles outlined in the abstract schema. A schema has place holders, variables, black boxes, that may be filled piecemeal as empirical evidence is found for the various components. The lack of an entity or activity or module to fill a role in a schema points to the need for further work" (p. S360).

Note that the aim of the scientists' work is not to show *that* something is true, but to explain *why* it is true. The explanation is provided as a description of the mechanism that produces the phenomenon, or at least a candidate mechanism that could produce it.

Russ (2006), building on the work of Darden and colleagues, argues that mechanistic reasoning is more effective than formal empirical investigation: "[I]t is by knowing what mechanisms act in the systems that scientists pare down the number of variables for further rigorous study. If scientists or students relied only on covariation data, they would be working forever testing infinite numbers of variables only a few of which would actually be important in the situation" (p. 43).

One might respond to these suggestions by pointing out that supporting language design choices is a different activity from developing scientific understanding. Surely one can decide whether A is better than B without having a scientific understanding of either, for which mechanisms would indeed be important. Here Russ provides two further relevant arguments.

First, Russ points out that mechanisms are important in distinguishing relevant correlations from irrelevant, drawing on Koslowski (1996). For example, lots of factors are correlated with crime in cities, including number of churches. Everyone recognizes that these correlations are not causal, just because there isn't a mechanism connecting the factors.

This might not seem relevant to language design, which doesn't much concern itself with correlations and their interpretation. But consider an A-B test of two language designs, that shows an advantage for A over B. Consideration of mechanisms might suggest that the observed advantage of A is due to some particular feature of the test task that would not be encountered often in practice. Similarly, consideration of mechanisms might suggest that had the test users possessed, or not possessed, some particular knowledge, B would have been better. (For an example of the impact of user knowledge on performance, also including interactions with task types, see Halasz and Moran, 1983.) The apparent clarity of the empirical test results would not stand up to mechanism-based critiques like these.

Finally, Russ argues that "Mechanistic reasoning is more helpful than formal empirical investigations for understanding novel situations"(p. 47). Her argument is parallel with our discussion of the difficulties posed by the tessellated design space, above:

> "When faced with new physical situations, it is unlikely that students will already have a store of covariation information collected under the exact same conditions with which to draw conclusions. Covariation information is narrow; it only gives insight into the precise case that generated it. Establishing that a particular variable is associated with a particular result in a very particular set of circumstances (a controlled experiment) does not help predict what will happen when that same variable is found in a different set of circumstances. Only in knowing the properties of a variable and the process by which a cause brings about an effect – the mechanism - can we know what that variable may or may not do in another situation" (p. 47).

Here again, Russ's focus on covariation shouldn't prevent us from recognizing the relevance of this conclusion for design work. Knowing that A outperforms B under particular test conditions can't give us confidence that it will do so under novel conditions, if we don't understand the mechanisms involved. Further, knowing that A outperforms B doesn't allow us to predict whether or not a novel design A' will outperform B, if we don't understand the mechanisms. As Landauer and Galotti (1984) say, commenting on confused empirical findings on command language design, "For completely practical purposes, only … theoretical understanding will make it possible to make good design decisions about new systems in advance" (p. 428).

Cognitive dimensions analysis can be seen as supplying a collection of mechanism schemata for explaining phenomena that occur when people use representations, in activities like programming. Applying cognitive dimensions analysis to a problem in designing a programming language has the same status as a biologist applying a mechanism schema to a problem in molecular biology. It has the same virtue of providing insight into *why* a design feature works or fails to work. It helps deal with the problems Russ identifies in formal empirical work.

Three other areas of thought bear on these matters, and can be mentioned briefly. First, some work in machine learning, following insights by Schank et al. (1986), has favored explanation-based generalization over similarity-based generalization (see e.g. Mitchell et al., 1986; Lewis, 1988), especially in situations in which generalizations from very limited data are required (see also Jones et al., 2006, for this distinction in psychological studies of generalization.) As argued earlier, language designers will very often be seeking to generalize in data-poor circumstances, for reasons that include novelty and contextual complexity. Second, workers in evidence-based medicine have recognized the difference between a prediction for a population and a prediction for an individual patient: "It may be hazardous to presume that the point estimate of risk derived from a population model represents the most accurate estimate for a given patient. … [D]irect measurement of subclinical disease (screening) affords far greater certainty regarding the personalized treatment of patients, whereas risk estimates often remain uncertain for patients. In conclusion, shifting our focus from prediction of events to detection of disease could improve personalized decision-making and outcomes." (McEvoy et al., 2014). Similarly, someone designing software for a very large audience, and able to collect performance data on a very wide scale, might feel that population-level A-B trials would give adequate design guidance. But the resulting design might prove to be quite bad for many particular users, in particular situations. Finally, in general HCI, claims analysis (Carroll and Kellogg, 1989; McCrickard, 2012) shares features with cognitive dimensions logic; see also Haynes et al. (2009).

4. Implications for PPIG

The PPIG community should be proud that cognitive dimensions analysis emerged from the work of people in its ranks, Thomas Green, Marian Petre, Alan Blackwell, and others. We should be skeptical of calls to replace its use with A-B trials or other quantitative methods that cannot cope with the complexity of the language design landscape. When results of A-B trials and similar studies are presented, we should diplomatically ask for the mechanisms that are involved to be described. Colleagues who present the results of such trials should be prepared to respond to this request, so that the generalizability of their results can be assessed. Thinking aloud studies, such as were used effectively by Halasz and Moran in their calculator work, or other ways of observing the activities of individual users in detail, as in Steven Clarke's videos, can be a good supplementary method for investigators who are doing empirical comparisons. Indeed, adding a cognitive dimensions analysis to an empirical comparison would also increase generalizability.

We should help our colleagues understand the value of mechanism-based analysis, given that for many it doesn't seem "scientific" or "technical" in ways they can readily recognize (see e.g. Moody, 2009). Studies that combine cognitive dimensions analysis with user observations, as was done so effectively by Clarke, could be a good way to do this.

Another, quite different, way could be to explore the relationship between cognitive dimensions and the state of the art in cognitive modeling, as represented by the work of the ACT-R and SOAR communities (http://act-r.psy.cmu.edu/, http://soar.eecs.umich.edu/). This could be seen as a way to clarify the role of cognitive dimensions as mechanism schemata, with roles to be filled by specific cognitive mechanisms described at a lower level. This could enhance their explanatory power, in line with Darden's description of scientific progress. See the related recommendations in Church and Mărășoiu, given at PPIG 2016.

Finally, we should seek to expand the repertoire of dimensions, even to include dimensions not properly cognitive. The tripartite picture of the design landscape shows not only people and machines but also purposes. We may be able to identify mechanism schemata that help people design for particular purposes, for example maintainability. One would suppose that failures of maintainability result from the action of mechanisms that could be identified. The discussion by Basman et al. (2016) at PPIG 2016 perhaps suggests what might be done. While such an effort would draw the Psychology of Programming Interest Group a little beyond the properly psychological, it could place PPIG's psychological work in a setting in which its value is more apparent to a wider audience.

5. Acknowledgements

I thank the Hanse-Wissenschaftskolleg, Delmenhorst, Germany, for support while preparing this essay. Emily Moore, Luke Church, Antranig Basman, and the students in CSCI 4830 in spring, 2014, provided useful ideas for which I am grateful.

6. References

Agate, S. J., & Drury, C. G. (1980). Electronic calculators: which notation is the better? *Applied Ergonomics*, **11**(1), 2-6.

Basman, A., Church, L., Klokmose, C., & Clark, C. (2016). Software and How it Lives On-Embedding Live Programs in the World Around Them. In *Proc. Psychology of Programming Interest Group Annual Workshop 2016* (Cambridge, England, 7-10 September, 2016).

Green, T., & Blackwell, A. (1998, October). Cognitive dimensions of information artefacts: a tutorial. In *BCS HCI Conference* (Vol. 98).

Card, S. K. (1979). A method for calculating performance times for users of interactive computing systems. In *Proc. International Conference on Cybernetics and Society* (pp. 653-658).

Carroll, J. M., & Kellogg, W. A. (1989). Artifact as theory-nexus: Hermeneutics meets theory-based design. In *Proc. SIGCHI 1989 Conference on Human Factors in Computing Systems* (pp. 7-14). ACM.

Church, L., & Mărășoiu, M. A fox not a hedgehog: What does PPIG know? In *Proc PPIG 2016 - 27th Annual Conference*, Cambridge, England.

Darden, L. (2002). Strategies for discovering mechanisms: Schema instantiation, modular subassembly, forward/backward chaining. *Philosophy of Science*, **69**(S3), S354-S365.

Green, T. R. (1989). Cognitive dimensions of notations. *People and computers V*, 443-460.

Green, T., & Blackwell, A. (1998, October). Cognitive dimensions of information artefacts: a tutorial. In *BCS HCI Conference* (Vol. 98).

Halasz, F. G., & Moran, T. P. (1983). Mental models and problem solving in using a calculator. In *Proc. SIGCHI 1983 conference on Human Factors in Computing Systems* (pp. 212-216). ACM.

Haynes, S. R., Carroll, J. M., Kannampallil, T. G., Xiao, L., & Bach, P. M. (2009). Design research as explanation: perceptions in the field. In *Proc. SIGCHI 2009 Conference on Human Factors in Computing Systems* (pp. 1121-1130). ACM.

Hoffman, E., Ma, P., See, J., Yong, C. K., Brand, J., & Poulton, M. (1994). Calculator logic: when and why is RPN superior to algebraic? *Applied ergonomics*, 25(5), 327-333.

Jones, M., Maddox, W. T., & Love, B. C. (2006). The role of similarity in generalization. In *Proc. 28th annual meeting of the cognitive science society* (pp. 405-410).

Kasprzyk, D. M., Drury, C. G., & Bialas, W. F. (1979). Human behaviour and performance in calculator use with Algebraic and Reverse Polish Notation. *Ergonomics*, **22**(9), 1011-1019.

Koslowski, B. (1996). *Theory and evidence: The development of scientific reasoning*. Cambridge, MA: MIT Press.

Kreifeldt, J. G. (1981). Hand calculator performance under interrupted operation. In *Proc. Human Factors and Ergonomics Society Annual Meeting* (Vol. 25, No. 1, pp. 329-332). SAGE Publications.

Landauer, T. K., & Galotti, K. M. (1984). What makes a difference when? Comments on Grudin and Barnard. **Human Factors**, 26(4), 423-429.

Lewis, C. (1988). Why and how to learn why: Analysis-based generalization of procedures. **Cognitive Science**, 12(2), 211– 256. doi: 10.1207/s15516709cog1202_3

Luff, M. (2009). Empirically investigating parallel programming paradigms: A null result. In Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU).

Machamer, P., Darden, L., & Craver, C. F. (2000). Thinking about mechanisms. *Philosophy of science*, **67**(1), 1-25.

McEvoy, J. W., Diamond, G. A., Detrano, R. C., Kaul, S., Blaha, M. J., Blumenthal, R. S., & Jones, S. R. (2014). Risk and the physics of clinical prediction. *The American journal of cardiology*, **113**(8), 1429-1435.

McCrickard, D. S. *Making Claims: Knowledge Design, Capture, and Sharing in HCI*. Synthesis Lectures on Human-Centered Informatics 5.3 (2012): 1-125.

Mitchell, T. M., Keller, R. M., & Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine learning*, 1(1), 47-80.

Moody, D. (2009). Theory development in visual language research: Beyond the cognitive dimensions of notations. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on* (pp. 151-154). IEEE.

National Center for Education Evaluation and Regional Assistance (2003) Identifying and Implementing Educational Practices Supported by Rigorous Evidence: A User Friendly Guide. Online at https://ies.ed.gov/ncee/pubs/evidence_based/evidence_based.asp.

Newell, A., & Card, S. K. (1985). The prospects for psychological science in human-computer interaction. *Human-computer interaction*, 1(3), 209-242.

Russ, R. S. (2006). A framework for recognizing mechanistic reasoning in student scientific inquiry (Doctoral dissertation, University of Maryland).

Schank, R. C., Collins, G. C., & Hunter, L. E. (1986). Transcending inductive category formation in learning. *Behavioral and Brain Sciences*, 9(4), 1469-1825.