Design opportunities for the psychology of programming after AI

Clayton Lewis
Emeritus Professor
University of Colorado Boulder
clayton.lewis@colorado.edu

Abstract

The advent of artificial intelligence coding tools calls for a shift in focus in the PPIG community from the psychology of writing programs toward the psychology of evaluating them. Using some simple (but real) examples in the domain of educational programming, this paper explores what approaches non-programmers might use to determine whether a program that has been given to them works as desired, including the roles of AI tools in this process, as well as in writing the code. The exploration suggests that cognitive dimensions are useful in understanding these matters, and that there are a number of research directions that may support the creation of programs that are easier for people to evaluate.

1. Introduction

This paper asks, what are the opportunities for PPIG when people don't have to write programs to have programs? Over the last few years, AI tools for programming have advanced rapidly. At PPIG 2023 I discussed a program like the one shown in Figure 1 (Lewis, 2023). GPT 4 could write such a program, but an iterative prompting process was needed. Now, Anthropic's Claude writes a working program in response to a single prompt:

> For a math class I'm teaching, I'd like a Web app that displays a 3x3 grid of coins, with buttons that flip all the coins in each row or column. I'd also need some way to set up any starting arrangement of coins, such as all heads. My idea is that my students can work on the problem of whether you can change all heads to all heads except for the center coin, using the legal operations.

In situations within the scope of such tools, the focus of attention shifts from "How can I write this program?" toward "How can I tell if this program I've been given does what I want?" What opportunities for the PPIG community does this shift entail? We can approach this question by considering how users can approach the evaluation of programs they are given.

For some programs, looking at the output is enough. If I just want my program to create an animated cat, I can just look at the cat I get, and decide if it is good enough. But more typically we want a program to behave in some correct way in the future, not just in some one current situation. How can we tell if a program we've been given will do that? Let's use the Nine Coins problem to explore this situation, or at least, to scratch the surface.

The Nine Coins program originated as an example of the use of the Boxer language, developed by Andy diSessa as a tool to support the development of computation as a tool for thought and expression in schools (diSessa & Abelson, 1986; Mason, 1995). One aim of Boxer is to allow students and teachers to develop tools for exploring ideas, perhaps abstract algebra in this case (the operations on the grid form a group). The idea is that, with the right sort of language, programs like this can be developed easily and spontaneously, in the course of classroom discussion. Students can then use the program to explore the operations on the grid, as in the example cited in the prompt, above.

It would be bad if the program didn't act correctly. Students might draw the wrong conclusion about the behavior of the operations on the coins, for example concluding that the configuration with one tail in the center can't be reached from all heads, when it actually can, or that it can't be reached when it can. This would lead to confusion, and wasted time, at best.

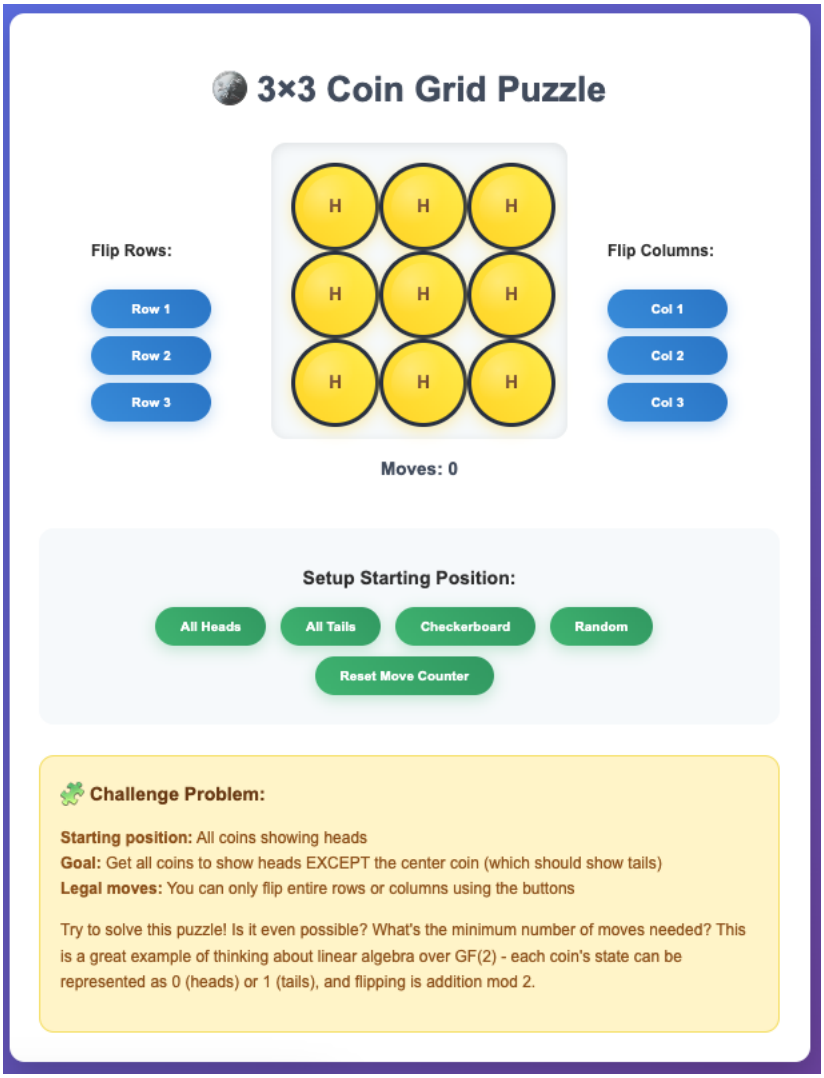

Figure 1 – Claude's Nine Coins program.

In the Boxer scenario, the class would have written the program, using the Boxer language. The process of writing the program would bring with it a sense of its correctness. But what if, as is now possible, they have simply asked for the program, and are given it, with no participation in writing it at all? What approaches could they take to gain confidence that the program will work?

Before digging into this question, let's note that the Boxer project has larger goals, and that programming in Boxer potentially has benefits that go beyond just getting a program that works. DiSessa argues that computing is a fundamental intellectual tool, and that mastering it should be considered a form of literacy on a plane with textual literacy or mathematical literacy (DiSessa, 2001). We won't consider this matter further here, beyond noting that developing these ideas in the context of AI tools would be an important PPIG contribution.

2. One way to evaluate the Nine Coins programs is to test it

An obvious approach is to press the buttons, and check what they do. It's very easy to do some of this, but not completely trivial to do it completely. There are 11 buttons, and one would have to be patient in trying each of them. But just testing each button doesn't give much assurance that the program will work reliably. It could be that certain sequences of button presses don't work correctly, or that a button would fail for just certain grid patterns.

In serious software testing, constructing an adequate collection of tests isn't easy. To give a fair level of

confidence a test suite should at least exercise each part of the code. But if one has no access to the code, or can't understand the code, as would be the case for our non-programming class, such test coverage isn't possible, without assistance.

2.1. AI tools can write and apply tests

One can ask Claude to test a Nine Coins program that it has provided; I've done this for a simpler form of the program than that shown in Figure 1, discussed later. The conversation can be seen at `https://claude.ai/public/artifacts/edfb6e97-78cf-4562-aed6-224935648221` (I'm including links to conversations with Claude for interested readers. But I'll try to describe the important points in the text, in part because these links may not work in perpetuity.)

Unfortunately what Claude says would not be easy to follow for a non-programmer. For example, it refers to "Boundary and Edge Case Tests"; what are those? Under that heading is "Boundary values - Tests first/last rows and columns (indices 0 and 2)". What are "indices"?

Claude did not spontaneously perform coverage analysis, but did when asked, as can be seen in the conversation. Unfortunately, the coverage display can't readily be linked to the actual code, using line labels (like "flipRow-3a") that aren't shown in the code listing users can see.

This wouldn't necessarily be a big problem. A non-programmer consumer could ask their tool to carry out this kind of evaluation, and just trust the results. In low stakes situations like Nine Coins that could well be adequate. In situations where more skepticism is warranted, could more support be offered, including to people who don't understand code? This is a design opportunity for the PPIG community: how can program tests and test results be informatively presented, for a range of audiences?

3. Another way to assess Nine Coins is to understand the code in a way that builds confidence

As mentioned earlier, part of the vision for the Boxer language is that people who write code can have some confidence based on their understanding of the code. Can people who didn't write code, and don't understand code, be helped to understand it?

Here is a conversation with Claude about its initial offering for Nine Coins: `https://claude.ai/share/520a561a-3cc0-4fad-9b08-ba066f03ae96`. Claude tries various ways to explain why the code can be trusted, none very easy to understand. For example, a piece is

> Grid Indexing: The 3×3 grid correctly maps to a linear array using index = row * 3 + col
>
> Position [0][0] → index 0, [0][1] → index 1, [0][2] → index 2

This excerpt refers to an aspect of the code that presents difficulty, the representation of the two-dimensional grid of coins with a linear array. Here is the code, discussed later in the conversation:

```
function flipRow(rowIndex) {
    for (let col = 0; col < 3; col++) {
        const index = rowIndex * 3 + col;
        grid[index] = !grid[index];
    }
}
```

This code works, but isn't easy for a non-programmer to follow, even leaving aside notational matters like how repeated operations are described. What does the index calculation in the third line have to do with the problem?

Another issue that can be seen here (by someone who understands programming) is the representation of a coin's state as a Boolean value. There are two aspects to this. First, the values in the grid are true and false, not heads and tails. That makes it hard to relate this code to the problem. Second, a non-

programmer would have no idea what the exclamation point means in this case. Even if they are told that it is negation, the relationship between this operation and flipping a coin has to be further explained.

In the course of the conversation (with quite a bit of prodding) Claude developed a simpler program that addresses these issues, shown in Figure 2 (see code in Listings 1 and 2).

Figure 2 – Nine Coins simplified by Claude.

This code is indeed easier to understand. The grid is 3 by 3. The representation of coin states is more transparent ("H" and "T"), and the repetition is spelled out, rather than expressed with loop logic. We can see in this conversation that Claude is capable of writing code that is easier to understand than its first offering. But it took iterative guidance from the user, identifying areas of difficulty, to get it to do it.

It's quite likely that prompts could be devised that would push Claude to produce simpler code, without requiring the user to make iterative requests for clarification. This isn't simple, though. This quick stab,

> For a math class I'm teaching, I'd like a Web app that displays a 3x3 grid of coins, with buttons that flip all the coins in each row or column. I'd also need some way to set up any starting arrangement of coins, such as all heads. My idea is that my students can work on the problem of whether you can change all heads to all heads except for the center coin, using the legal operations. I'll want to be able to understand the code well enough that I'll be confident that all the buttons will do exactly what they are supposed to do. And I'm not a programmer. Can you make the code so simple that you can easily explain it to me?

was not effective in producing code as simple as what was obtained in the conversation above by iterative requests. For example, Claude used Booleans in this "simple" code to represent the coins. So here's another design opportunity for PPIG: Can we develop generic prompting language that can get coding tools to produce really simple, easy to understand code? Packing examples of what's wanted into the prompt could be an approach.

4. What could make code really much easier to understand?

What we've seen Claude do seems to be just nibbling around the edges of understandability. A problem that's been recognized by the PPIG community, that Claude hasn't touched, is what Steve Draper calls the pumping problem (Draper, 1986). In the code in Table 1, consider the function show(). There's the question of what's being said there: What is "+" in this context? What is "InnerHTML"? There's also a more basic question: why is this code needed at all? Isn't updating the grid of coins all the program needs to do?

As programmers know, no, it isn't. In most programming systems, the entities being manipulated can't be seen. Rather, code is needed to pump information from inside the computer to the outside (other pumping code is needed to take information in.) As Draper observed, spreadsheets avoid pumping code: data in a spreadsheet are visible, without the user having to do anything to make them so. The success

Listing 1 – Code for simplified Nine Coins program.

```
<!DOCTYPE html>
<html>
<head>
    <title>Simple Coin Grid</title>
</head>
<body>
    <h1>3x3 Coin Grid</h1>
    <div id="display" style="font-family: monospace; font-size: 24px;
         line-height: 1.5;">
        <!-- Grid shows here -->
    </div>
    <p>
        <button onclick="flipRow(0)">Flip Row 1</button>
        <button onclick="flipRow(1)">Flip Row 2</button>
        <button onclick="flipRow(2)">Flip Row 3</button>
    </p>
    <p>
        <button onclick="flipCol(0)">Flip Col 1</button>
        <button onclick="flipCol(1)">Flip Col 2</button>
        <button onclick="flipCol(2)">Flip Col 3</button>
    </p>
    <p>
        <button onclick="allHeads()">All Heads</button>
    </p>
    <script>
        // The grid
        let grid = [
            ["H", "H", "H"],
            ["H", "H", "H"],
            ["H", "H", "H"]
        ];

        // Show the grid
        function show() {
            let text = "";
            text += grid[0][0] + " " + grid[0][1] + " " + grid[0][2]
                 + "<br>";
            text += grid[1][0] + " " + grid[1][1] + " " + grid[1][2]
                 + "<br>";
            text += grid[2][0] + " " + grid[2][1] + " " + grid[2][2];
            document.getElementById('display').innerHTML = text;
        }
```

Listing 2 – Code for simplified Nine Coins program, continued.

```
        // Flip a row
        function flipRow(row) {
            if (grid[row][0] === "H") { grid[row][0] = "T"; }
            else { grid[row][0] = "H"; }
            if (grid[row][1] === "H") { grid[row][1] = "T"; }
            else { grid[row][1] = "H"; }
            if (grid[row][2] === "H") { grid[row][2] = "T"; }
            else { grid[row][2] = "H"; }
            show();
        }

        // Flip a column
        function flipCol(col) {
            if (grid[0][col] === "H") { grid[0][col] = "T"; }
            else { grid[0][col] = "H"; }
            if (grid[1][col] === "H") { grid[1][col] = "T"; }
            else { grid[1][col] = "H"; }
            if (grid[2][col] === "H") { grid[2][col] = "T"; }
            else { grid[2][col] = "H"; }
            show();
        }

        // Reset to all heads
        function allHeads() {
            grid[0][0] = "H"; grid[0][1] = "H"; grid[0][2] = "H";
            grid[1][0] = "H"; grid[1][1] = "H"; grid[1][2] = "H";
            grid[2][0] = "H"; grid[2][1] = "H"; grid[2][2] = "H";
            show();
        }

        // Start by showing the grid
        show();
    </script>
</body>
</html>
```

of spreadsheets suggests that changing the stuff programs work with can help with understanding. Code that works with stuff that's visible by default could be easier to understand.

4.1. Understanding more about understanding could help

To go farther, it would be useful to have more insight into how understanding works. Existing conceptions of understanding have focused on knowledge structures, and relationships among them; see for example (Gentner, 1983). Another idea is that understanding includes identifying causal mechanisms at work in a situation (Lewis, 1988; Russ, Coffey, Hammer, & Hutchison, 2009).

The unexpected success of Large Language Models, that seem to function largely without explicit provision for structures or causal mechanisms, suggests that mental processes result from the action of a large collection of small predictive regularities, linked by analogical relationships. This idea is explored in (Lewis, 2025).

A plausible reconception of understanding is that it is itself a predictive process. One has understood a novel situation when one can predict what will happen in it, as different events occur. Regularities observed in one situation can be used to make predictions in another, when analogies link the situations.

4.2. Phenomena in physics learning illustrate this process of understanding

Andy diSessa, in addition to his work on the Boxer system, has made substantial contributions to our understanding of how understanding develops in physics. One of these contributions is the idea of phenomenological primitive, or p-prim (DiSessa, 1993). For example, Ohm's p-prim describes a collection of analogies that link the predictive regularity "more voltage and same resistance gives more current" to the regularity "more push and same friction gives more motion", in a domain of pushing things on a surface (and many more, in the same domains, and other domains.) This constellation of some push, and some inhibition, and some result shows up in many situations, and can readily be understood.

Jim Minstrell, in earlier work (Minstrell, 1982), showed how students could understand how it is that a table, seemingly a passive object, can exert force, when an object is lying on it. By shining a laser beam off a mirror on the table top, Minstrell was able to show that the table deforms very slightly when an object is placed on it. That allowed the students to see that the table acts like a spring, something they generally accept as capable of exerting force. Minstrell's approach builds understanding of force exerted by a table from understanding of springs.

4.3. This view of understanding implicates the cognitive dimension, closeness of mapping

We can apply these ideas to understanding code, using the Nine Coins program. Cognitive dimensions analysis (Green, 1989; Blackwell et al., 2001), among many other references, highlights the importance of closeness of mapping, how easy it is to connect a representation to a user's idea of what it represents. Focusing on the code that flips the coins, we can see that both versions of the program pose challenges. In the original program, the Boolean negation operation was used, something non-programmers will almost certainly have little or nothing to connect to:

```
grid[index] = !grid[index];
```

There's a gap opened up in that program between the familiar domain of coins, with their natural behavior, and the unfamiliar domain of Boolean values and operations on them. As discussed earlier, there is also a mapping gap between the two-dimensional grid of coins, and the one-dimensional list of Boolean values.

The simplified program, in Figure 2, uses a closer mapping, between "H" or "T" and the heads or tails of coins. But there is still trouble, as looking at the code shows. Here's the code for flipping one of the coins in the grid:

```
if (grid[1][col] === "H") { grid[1][col] = "T"; } else {
    grid[1][col] = "H"; }
```

Even leaving aside the cryptic notation, such as the distinction between "===" and "=", this seems an

odd representation of the idea of flipping. Flipping doesn't seem to require any conditional logic, but this representation of it does.

These two code snippets illustrate, in different ways, that understanding code can take people beyond things they already know about. With programming as it is, people have to spend weeks or longer to be able to reason about code. Could that be avoided? There's another PPIG design opportunity here: Could programs be constructed from materials whose behaviors are closer to behaviors non-programmers are already familiar with?

5. Capitalizing on knowledge of the behavior of physical objects in a programming system

A domain people have a lot of experience with is physical objects. Making computational objects more like physical objects could have advantages, in making their behavior easier to understand.

One advantage is that physical objects don't require pumping code, while common computational objects do. Pumping code poses multiple challenges for understanding. One has to understand the code itself, for example the concatenation of text in the show() function in Listing 1. But also one has to understand why pumping code is needed at all. That is, one has to understand all of what the code is doing, how it is doing it, and why it is doing it. If computational objects were more like physical objects, none of this would be needed. Values in spreadsheets show that doing without pumping is possible, at least to some extent.

It is possible, though not easy, to write Nine Coins as a spreadsheet. Current spreadsheets push one into a scripting environment that has its own challenges, inherited from the scripting language. For example, it's not simple to access the value of a cell in the script.

Claude can navigate these challenges, just as it can write the code for the Nine Coins versions we've been looking at. But the code isn't any easier to understand. In particular, flipping a coin still involves a conditional construction, much like the one in Claude's simple program (Listings 1 and 2).

The trouble is that spreadsheet values share only some of the behaviors of physical objects. In particular, a physical object like a wooden block can be flipped, but a spreadsheet value cannot.

5.1. A fantasy coding system would have computational objects that obey the naive physics of real objects

Here are some of the familiar affordances of common physical objects, like children's blocks:

- They are always visible.
- They don't move or change on their own.
- They can't easily be modified, but they can be rotated.
- They exist in places, and can be referred to by location.
- Groups of locations can be referred to (grids, rows, columns, etc.)

A coding tool for working with such objects would write scripts using natural references ("flip all the coins row 1", "turn all the coins in the grid to heads") and attach them to buttons. The script attached to a button could be shown by the tool as part of an explanation, or displayed by the user. It's easy to see how Nine Coins could be implemented in a system like this, and that it would be easy to understand, and have confidence in, the implementation.

(Code that controls the appearance of objects would not normally be shown. We've not discussed this, but such code is of course there. In the simple program in Table 1 there's styling on the where the coins are shown. This is potentially a tricky matter for our fantasy system, because physical objects don't have styles.)

5.2. A more complex example for the fantasy system

The possibilities of this fantasy system can be dramatized with another example from (Mason, 1995):

> A cube is placed on each square of a chessboard. The faces of the cubes are congruent to the squares of the board. Each of the cubes has at least one black face. We are allowed to rotate a row or column of cubes about its axis. Prove that by using these operations, we can always arrange the cubes so that the entire top side is black.

As discussed in (Lewis, 2022), it's possible to solve this problem fairly easily (for a programmer, using an AI tool) using three numbers to represent the orientation of each cube. One of the numbers indicates the upper face of the cube, and the other two indicate two other faces. This allows the rotations to be simulated quite simply. (As discussed in the reference, two face numbers would be enough, but one has to assume the handedness of the cube numbering to make that work.)

A solution in the fantasy system would be much easier to understand, though. The computational cubes would be rotated in the same way as real cubes, with no representational trickery required.

5.3. Another example is the swapping problem

Novices often have trouble swapping the value of two variables in common programming languages. As programmers learn, a common attempt,

```
X = Y
Y = X
```

fails:

```
(start with X contains 2, Y contains 3)
X=Y now X contains 3, Y contains 3
Y=X now still X contains 3 and Y contains 3
```

To avoid this one can use a third variable, like this:

```
(start with X contains 2, Y contains 3, T contains anything)
T=X X contains 2, Y contains 3, T contains 2
X=Y X contains 3, Y contains 3, T contains 2
Y=T X contains 3, Y contains 2, T contains 2
```

The trouble with the first attempt is that with ordinary computational objects, the first statement destroys the value of X. That doesn't happen with physical objects. If values are represented by wooden blocks, I can move them around without fear of destruction.

But an emergent problem with physical objects is that when I move the block in location Y to location X, now I have two objects there. How do I move just the original block to location Y? That can be expressed in natural reference as "move the block that was in X to location Y".

Another swap technique, that works only for numbers, highlights another failure of physical intuition:

```
(start with X contains 2, Y contains 3)
X=X+Y X now contains 5; Y contains 3
Y=X-Y X contains 5; Y contains 2
X=X-Y X contains 3; Y contains 2
```

This works, but it is far from easy to see that it does. On the face of things, after the first step, the value 2 has completely disappeared!

In the fantasy block system the solution would be something like

```
Move the block in Y to X
Move the old block in X to Y
```

or even just

```
Swap the blocks in X and Y
```

Note: Some programming languages have added a swap operation to deal with this problem. For example, in Python one can form and assign tuples, using commas:

```
X,Y = Y,X
```

But then this notation has to be explained.

One might think that this little problem, swapping, is too simple to warrant attention. But a quick Google search for "swapping two variables" will show a great many people asking about it, and posting suggestions.

5.4. The intent of the fantasy project is not to make it easier to write programs, but to make them easier to explain

It would likely be easier for a non-programmer to write the natural references that are imagined in the fantasy system, than references in current languages. But that's not the point. Rather, the idea is that they don't have to write them at all. The coding tool writes the references. The coding tool is also responsible for ensuring that the actual implementation code it writes (in some conventional language) honors the explanatory description.

This approach follows a suggestion of Antranig Basman that AI tools open up the design space for communities like PPIG. That's because the tools can do the work of translating from novel representations to existing ones. Thus there's another PPIG Design opportunity: make something like this fantasy coding system actually work, leveraging the resources of AI coding tools.

Basman's work contains many other relevant reflections on the stuff of computing, and how much computational stuff suffers in comparison to physical stuff (Basman, 2016, 2017). In the same vein, Lida Kindersley, in a contribution to the joint workshop of PPIG and Art Workers Guild in 2020 (`https://ppigattheartworkersguild2018.wordpress.com/2020/12/01/postcard-from-lida-kindersley/`) reported that she found working as an artist/craftsperson in stone was "much more fun, much more direct, and much more surprising (emphasis in the original)" than working with computational stuff.

6. This fantasy project likely raises more questions than it answers

These ideas only deal with some pretty simple issues with data. How would a system like this cope with the complexity of control structures?

The Forms2 system (Ambler & Burnett, 1989) uses an approach in which different layers of a recursive process are shown simultaneously, rather than earlier values being replaced by later ones. That involves copying, not a simple process for physical objects, but the visibility in the scheme could help people understand what's happening.

How much "real programming" could be done this way? That remains to be seen. Computer scientists were initially baffled that something as obviously limited as spreadsheets could be interesting to anybody. But spreadsheets have of course been wildly successful.

More generally, sacrificing expressiveness for other values can be worthwhile. (Kell, 2009) notes that programs in languages that aren't Turing complete can be much easier to reason about than others. This of course is quite relevant to the understanding problem we are considering.

7. Applying the cognitive dimensions approach to these design challenges

Little if any extension is needed to accommodate the design ideas we've considered so far. Rather, one needs only to consider how existing dimensions apply to representations intended to support code understanding. Since the key suggestion here is that understanding requires mapping what code does to intuitions the user already has, this is mostly about closeness of mapping. There's the (minor) difference

that the intuitions one wants to map to needn't be specifically in the user's problem domain, as is looked for in other applications of the closeness of mapping dimension. They just have to be things the user is used to thinking about.

An issue we haven't discussed, because the examples we've considered have been so simple, is understanding systems that likely have to be understood in parts. For example, a program that uses a complicated function can be understood by separating the problem of understanding how the function works, from the problem of understanding the system as a whole.

As in this example, such decompositions often themselves use technical apparatus that has to be understood itself, like parameters and different forms of argument passing. Perhaps a kind of cognitive dimension could reward designs for which the means of combining parts are themselves simple. While aspects of this problem may be covered in the existing system of dimensions, for example "hard mental operations", and "progressive evaluation", some more specific attention might be given to this aspect of understanding.

An example of an innovation that could be explored in this connection is Lanier's phenotropic programming (Lanier, 2002, 2003)(Lanier, 2002, 2003; see also (Lewis, 2018). In this conception, system components communicate by measurement of attributes of one another, rather than by a protocol like argument passing. Possibly this approach engages users' intuition better than protocols do.

So here are two more PPIG design opportunities: (1) Are there other concerns in code understanding and explanation that should be addressed by cognitive dimensions? (2) Are there ways of coupling the components of complex systems that are easier to understand than existing schemes?

8. Did I ask my AI tool the right question?

There's one more aspect of the program evaluation problem that should be addressed. I may be given a program that implements exactly what I asked for. It would pass any tests my tooling would devise, and an explanation of it would make me confident that the code does what I asked for. But it could be wrong, because I didn't ask for the right thing.

I experienced exactly this problem, in using an AI tool to create an agent-based simulation of similarity-based imitation. I wanted abstract agents that would carry out actions in such a way that they would imitate the neighbor that was most similar to themselves, based on their history of actions. The AI tool I was using easily wrote the simulation for me, but the results didn't seem right. I spent a lot of time (and used my knowledge of programming) trying to pin down the mistake. I assumed the tool had gotten something wrong. Eventually I realized that the program was exactly what I had asked for, but I had asked for the wrong thing. I told the tool to use cosine similarity in comparing action histories, whereas I should have used a different measure. Cosine similarity evaluates the angle between two vectors, and quite different vectors can have zero angle between them, for example [1,1,1] is parallel to [9,9,9].

In a later interaction I asked Claude whether it thought the cosine measure was appropriate: `https://claude.ai/share/c69ce235-826e-4a03-812f-70597b57f31b`. It provided a helpful discussion that raised the issue I had found, though it did not assert that I was wrong to use cosine. In another interaction I asked Claude to create the simulation, without specifying the similarity measure. Claude used a better measure, but it didn't comment on any connection between my goals and that measure. So there's no indication that it reasoned about my goals, other than in a very general way, in making its choice.

There's another PPIG design opportunity here. Is there a way to prompt AI tools that makes it more likely that the tool will provide feedback on what was asked, rather than just complying?

9. Conclusion

We've discussed only a tiny corner of the wide landscape of program evaluation. In an earlier PPIG paper (Lewis, 2017) I argued that the reason that the cognitive dimensions idea is so powerful, and such

"rigorous" methods as Randomized Controlled Trials are so unhelpful, is that programming, as a design space, is cut through by an enormous number of consequential distinctions. For example programs like Nine Coins ought actually to work, but it's not worth expending enormous effort to be sure they do. For the programs that control a car's braking system, it is worth making that effort. So completely different approaches, such as formal methods, and technically sophisticated test coverage disciplines, are called for in such situations. How to make techniques like these work well is a different design opportunity for PPIG. It's also one where AI tools will play a big role, as they can in the work discussed here.

10. Acknowledgements

I thank Antranig Basman and Owen Lewis for many important ideas. I thank the PPIG community, and the Hanse-Wissenschaftskolleg, Delmenhorst, Germany, for intellectual support over many years.

11. References

Ambler, A. L., & Burnett, M. M. (1989). Visual languages and the conflict between single assignment and iteration. In *1989 ieee workshop on visual languages* (pp. 138–139).

Basman, A. (2016). Building software is not a craft. In *Proceedings of the psychology of programming interest group* (p. 142).

Basman, A. (2017). If what we made were real. In *Proceedings of the psychology of programming interest group.*

Blackwell, A. F., Britton, C., Cox, A., Green, T. R., Gurr, C. A., Kadoda, G., … Young, R. M. (2001). Cognitive dimensions of notations: Design tools for cognitive technology. In *Cognitive technology: Instruments of mind: 4th international conference, ct 2001 coventry, uk, august 6–9, 2001 proceedings* (pp. 325–341). doi: 10.1007/3-540-44617-6_31

DiSessa, A. A. (1993). Toward an epistemology of physics. *Cognition and instruction*, *10*(2-3), 105–225.

DiSessa, A. A. (2001). *Changing minds: Computers, learning, and literacy*. Mit Press.

diSessa, A. A., & Abelson, H. (1986). Boxer: A reconstructible computational medium. *Communications of the ACM*, *29*(9), 859–868.

Draper, S. W. (1986). Display managers as the basis for user-machine communication. In *User centered system design* (pp. 339–352). CRC Press.

Gentner, D. (1983). Structure-mapping: A theoretical framework for analogy. *Cognitive science*, *7*(2), 155–170.

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

Kell, S. (2009). The mythical matched modules: overcoming the tyranny of inflexible software construction. In *Proceedings of the 24th acm sigplan conference companion on object oriented programming systems languages and applications* (pp. 881–888).

Lanier, J. (2002). The complexity ceiling. In J. Brockman (Ed.), *The next fifty years: Science in the first half of the twenty-first century* (pp. 216–229). Vintage.

Lanier, J. (2003, November). *Why gordian software has convinced me to believe in the reality of cats and apples.* edge.org. Retrieved from `https://www.edge.org/conversation/jaron_lanier-why-gordian-software-has-convinced-me-to-believe-in-the-reality-of-cats`

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

Lewis, C. (2017). Methods in user oriented design of programming languages. In *Proceedings of the 28th annual conference of the psychology of programming interest group (ppig 2017).*

Lewis, C. (2018). Phenotropic programming? In *Proceedings of the 29th annual conference of the psychology of programming interest group (ppig 2018).*

Lewis, C. (2022). Automatic programming and education. In *Companion proceedings of the 6th international conference on the art, science, and engineering of programming* (pp. 70–80).

Lewis, C. (2023). Large language models and the psychology of programming. In *Proceedings of the 34th annual conference of the psychology of programming interest group (ppig 2023)* (pp. 77–95).

Lewis, C. (2025). Artificial psychology. *Synthesis Lectures on Human-Centered Informatics*.

Mason, J. (1995). Exploring the sketch metaphor for presenting mathematics using boxer. In *Computers and exploratory learning* (pp. 383–398). Springer.

Minstrell, J. (1982). Explaining the "at rest" condition of an object. *The physics teacher*, *20*(1), 10–14.

Russ, R. S., Coffey, J. E., Hammer, D., & Hutchison, P. (2009). Making classroom assessment more accountable to scientific reasoning: A case for attending to mechanistic thinking. *Science Education*, *93*(5), 875–891.