Six Learning Barriers in End-User Programming Systems

Andrew J. Ko, Brad A. Myers, and Htet Htet Aung
Human-Computer Interaction Institute
Carnegie Mellon University, Pittsburgh, PA 15213 USA
ajko@cmu.edu, bam+@cs.cmu.edu, hha@cs.cmu.edu

Abstract

As programming skills increase in demand and utility, the learnability of end-user programming systems is of utmost importance. However, research on learning barriers in programming systems has primarily focused on languages, overlooking potential barriers in the environment and accompanying libraries. To address this, a study of beginning programmers learning Visual Basic.NET was performed. This identified six types of barriers: design, selection, coordination, use, understanding, and information. These barriers inspire a new metaphor of computation, which provides a more learner-centric view of programming system design.

1. Introduction

According to the U.S. Department of Labor, by 2012 30% of new jobs and nearly 8% of *all* U.S. jobs could require programming skills [1]. This is a dramatic shift for a skill that less than a million people had 10 years ago. Now, an increasing number of end-user programmers control manufacturing robots, create spreadsheets, and design interactive prototypes.

Yet, for such growth to occur, millions of aspiring end-user programmers must overcome substantial learning barriers in programming systems. Do we know enough about these barriers to design systems that help these individuals? We know much about the learning barriers in programming *languages* [11], but little about the rest of a programming system, which includes its environment (the editor, debugger, help, etc.) and accompanying libraries. What barriers do these parts of a programming system pose, if any?

In this paper, we answer this question both empirically and metaphorically. We begin by describing a study of Visual Basic.NET (VB), which identified six types of learning barriers. We then discuss several implications and describe a new metaphor of computation that facilitates a more learner-centric view of programming system design.

2. Prior Research on Learning Barriers

One way to understand learning barriers is to study the learner. For example, imagine Jill, a user interface designer who just began learning VB. Shortly after starting, she realizes that she must learn about event handlers to proceed. This poses a potential learning barrier. From an attention-investment perspective [2], she will weigh the cost, risk, and reward of overcoming the barrier, and if the risk of failure outweighs the reward, she may abandon VB for other tools.

Jill may also decide that progress is worth the risk of failure. We have proposed a framework that suggests she will make several *simplifying assumptions* about VB's language, environment, and libraries in trying to acquire the necessary knowledge [8]. If her assumptions are valid with respect to the programming system, she will make progress. If her assumptions are invalid—what we call *knowledge breakdowns*—she is likely to make an error. Within this framework, we define *learning barriers* as aspects of a programming system or problem that are prone to such invalid assumptions. These concepts are depicted in Figure 1.

Given these definitions, what aspects of programming systems might pose learning barriers? Research has explicitly identified several aspects of programming *languages* that are prone to invalid assumptions, including conditionals, Boolean operators, loops and data structures [12, 15]. Others have found that the task-specificity of language constructs influences learner's assumptions [4].

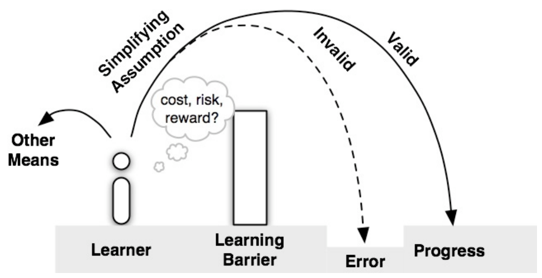

Figure 1. In overcoming barriers, learners risk making invalid assumptions that often lead to error.

However, our knowledge of potential barriers in the *rest* of the programming system is still vague. For example, Pane's efforts to identify usability issues in programming systems led to many heuristics [11]:

- Use signaling to highlight important information.
- Support incremental testing and feedback.
- Choose an appropriate computational metaphor.
- Help detect, diagnose, and recover from errors.
- Provide guiding knowledge through online help.
- Support recognition rather than recall.

Although these heuristics address importance concerns such as visibility and feedback, they fail to identify *particular* barriers in programming systems. If we could identify and define these potential barriers, we might discover *when*, *why*, and *for what* concerns such as visibility and feedback are relevant. This is precisely our goal in the remainder of this paper.

3. A Study of Visual Basic.NET 2003

To discover learning barriers in end-user programming systems beyond just those in languages, we observed 40 non-programmers learning to use Visual Basic.NET in a course called *Programming Usable Interfaces* (http://www.bam.hcii.cmu.edu/pui). We chose to study VB because it is aimed at end-user programmers and offers similar capabilities to other end-user programming systems. It also has the added complexities of a compiled language, the Visual Studio environment and the object-oriented .NET framework.

Our methodology sampled incidents of learners reaching *insurmountable* barriers: properties of VB or a programming problem that the learner could not understand despite considerable effort. Learners were told that if they were stuck they could consult an oracle (the experimenters) for guidance. When learners sought advice via e-mail or in a public lab, they were asked to report (1) what they were stuck on, (2) how they became stuck, and (3) how they tried to get "unstuck." The oracles then helped learners overcome their barrier. Learners worked on the tasks in Table 1 over 5 weeks. None of the learners who sought advice had taken more than one programming course.

We sampled 74 insurmountable barriers overall, but found that many of these were reached as a result of invalid assumptions that learners had made to overcome earlier barriers (as shown in Figure 2). By including these intermediate barriers, our final set included 130 barriers. We analyzed each with respect to learners' activities and use of VB, identifying six distinct categories. Two of the authors independently classified each barrier, attaining 94% agreement.

1. Create a form that computes the average of 3 numbers in text fields.
2. Fix a form so that it sorts the names in the list reverse-alphabetically.
3. Write a program that is impressive in its utility or entertainment value.
4. Create a form with a chain of interaction using all of the VB widgets.
5. Design an alarm clock that can be set to ring at a certain time.
6. Make a simulation that shows 3 elevators' directions and floors.
7. Design a copy machine interface that supports collating and stapling.

Table 1. The seven VB learning tasks.

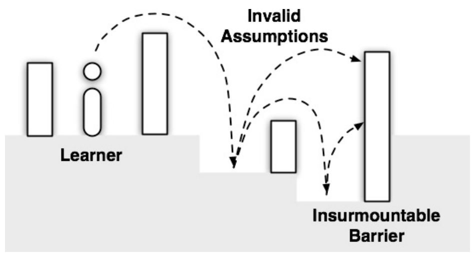

Figure 2. Learning barriers overcome with invalid assumptions often led to insurmountable barriers of a different type.

4. Six Learning Barriers

The six barriers we identified are: *design*, *selection*, *coordination*, *use*, *understanding*, and *information* barriers. Our definitions of these barriers rely on the concept of a *programming interface*: any element of a programming system's language or accompanying libraries that can be used to achieve some behavior. These include language constructs such as loops and operators, and library calls such as animations and math routines. Programming interfaces should not be confused with the user interfaces in an environment (compilers, editors, menus, etc). We will distinguish the two in our discussions.

4.1 Design Barriers

I don't know what I want the computer to do...

Design barriers (4 of 130) are inherent cognitive difficulties of a programming problem, separate from the notation used to represent a solution (i.e., words, diagrams, code). Several problems posed design barriers, including sorting, communication between forms, conditional logic, and event concurrency.

Half of the design barriers were insurmountable (2 of 4) because solutions to a problem were difficult to visualize. For example, a learner working on task 2 (see Table 1) was unable to conceive of a systematic way to sort names. Her best solution was "Just keep moving the names until it looks right!" Learners who were able to conceive of a sorting algorithm made invalid assumptions about their solution. For example, one learner successfully tested one cycle of her algorithm on a single data set on paper, and believed it to be correct. When her algorithm failed, she faced the insurmountable understanding barrier of determining what her algorithm did and did not do at runtime.

4.2 Selection Barriers

I think I know what I want the computer to do, but I don't know what to use...

Selection barriers (13 of 130) are properties of an environment's facilities for finding what programming interfaces are available and which can be used to achieve a particular behavior. These emerged when learners could not determine which programming interfaces were capable of a particular behavior.

Half of the selection barriers were insurmountable (6 of 13). Many learners faced selection barriers in task 5 in trying to get their program to keep time. Some tried using the help system, but could not guess which keywords to use. If they happened to find a relevant article, they were unable to understand the description of VB's timing abilities. Many learners overcame selection barriers by using their peers' timing code as examples, but faced insurmountable *use* and *coordination barriers* in adapting them.

4.3 Coordination Barriers

I think I know what things to use, but I don't know how to make them work together...

Coordination barriers (25 of 130) are a programming system's limits on how programming interfaces in its language and libraries can be combined to achieve complex behaviors—what one learner called "the invisible rules." Learners encountered these when they knew what set of interfaces could achieve a behavior, but did not know how to coordinating them.

Most coordination barriers were overcome with invalid assumptions (20 of 25). For example, learners correctly assumed that inter-form communication involved creating a new form programmatically and accessing its data (in VB a "form" is a window). However, most made invalid assumptions about how to access data and tried to "pull" values from the new form instead of "pushing" values to the old form. Because form controls are inaccessible if their form is not visible, "pulling" led to runtime exceptions.

Learners also overcame coordination barriers by finding examples that revealed VB's invisible rules. However, as with selection barriers, they faced *use* and *coordination barriers* adapting them to their needs.

4.4 Use Barriers

I think I know what to use, but I don't know how to use it...

Use barriers (36 of 130) are properties of a programming interface that obscure (1) in what ways it can be used, (2) how to use it, and (3) what effect such uses will have. These arose when learners knew what interface to use, but were misled by these obscurities.

Half of the use barriers were insurmountable (17 of 36), often because a programming interface did not indicate in what ways it could be used. For example, task 4 required learners to make a *Label* interactive, but many did not know that a *Label* could respond to mouse events. Some overcame these use barriers by using VB's facilities for obtaining a list of an object's methods. However, learners made invalid assumptions about how to use the methods or what effects they would have, passing syntactically correct but semantically *incorrect* parameters (also use barriers).

Use barriers were also insurmountable when they involved syntax. For example, learners could not determine how to declare or initialize arrays; when they guessed, they made invalid assumptions, and encountered insurmountable *understanding barriers* in determining the meaning of the resulting syntax errors.

4.5 Understanding Barriers

I thought I knew how to use this, but it didn't do what I expected...

Understanding barriers (38 of 130) are properties of a program's *external* behavior (including compile- and run-time errors) that obscure what a program did or did not do at compile or runtime. These emerged when learners could not evaluate their program's behavior relative to their expectations.

Most understanding barriers were insurmountable (34 of 38). Compile-time errors were insurmountable when learners could not determine what parts of their code were deemed right or wrong by the compiler, based on its error message. For example, when learners wrote a function call without a '=', they received the error message "*expected: =*". Learners faced an understanding barrier of determining if and where the '=' should be placed, and why it was "expected."

Runtime-errors and other unexpected behavior were insurmountable when they obscured what did or did not happen at runtime. For example, some learners wanted to pass data between forms, but did not know how to create references to each. To overcome this use barrier, they assumed that they could instantiate a form of the appropriate type in the *Form_Load* event of each form, not knowing this would cause infinite recursion and a stack overflow exception. Most learners did not associate the exception with their earlier assumption, because it did not suggest a relationship to their code.

In other cases, learners expected a behavior that did not occur. For example, many learners created a *Timer* object, assuming that it would start counting at runtime, when it was in fact disabled by default. When their label's text did not update as expected, they overlooked their assumption, and as a result, could not imagine what prevented the label from updating. Most assumed that their update code was incorrect, and rewrote it. Of course, this led directly to the same understanding barrier.

4.6 Information Barriers

I think I know why it didn't do what I expected, but I don't know how to check...

Information barriers (14 of 130) are properties of an environment that make it difficult to acquire information about a program's *internal* behavior (i.e., a variable's value, what calls what). These arose when learners had a hypothesis about their program's internal behavior, but were unable to find or use the environment's facilities to test their hypothesis.

Many information barriers were insurmountable (10 of 14) because the places to search for appropriate tools were numerous, or it was unclear how to use a tool. For example, many learners accidentally closed VB's property panel and could not determine how to redisplay it. Some learners caused null pointer exceptions, but did not notice that the exception dialog contained a link to the code responsible.

Some learners overcame information barriers by assuming something about their program's behavior. For example, when learners could not find the code that caused a null pointer exception, they deleted all of their recently modified code, confident that part of it must be guilty. When learners encountered barriers in using VB's debugger, rather than overcome them, they abandoned the debugger and simply guessed which statement was to blame.

5. Discussion

5.1 The Gulfs of Execution and Evaluation

The barriers share characteristics of Norman's *gulf of execution* (the difference between users' intentions and the available actions) and *gulf of evaluation* (the effort of deciding if expectations have been met) [10]. Three barriers pose gulfs of execution exclusively:

- *Design*: mapping a desired program behavior to an abstract description of a solution.
- *Coordination*: mapping a desired behavior to a computational pattern that obeys "invisible rules."
- *Use*: mapping a desired behavior to a programming interface's available parameters.

Two pose gulfs of execution *and* evaluation:

- *Selection*: mapping a behavior to appropriate search terms for use in help or web search engines, and interpreting the relevance of the results.
- *Information*: mapping a hypothesis about a program to the environment's available tools, and interpreting the tool's feedback.

Understanding barriers pose gulfs of evaluation exclusively, in interpreting the external behavior of a program to determine what it accomplished at runtime.

We can adapt Norman's recommendations on bridging gulfs of execution and evaluation to programming system design. For example, Norman recommends bridging gulfs of execution by establishing visible constraints on what actions are possible. For coordination barriers, this might involve a more explicit representation of a system's "invisible rules." To overcome gulfs of evaluation, Norman recommends that a system's state be accessible and understandable relative to users' expectations. For understanding barriers, this suggest a tool such as the Whyline [7], which provides explains what behaviors a program did and did not do at runtime.

5.2 How Are the Learning Barriers Related?

The graph in Figure 3 reveals common paths of failure in learning VB. The edges show the percent of each type of barrier that was overcome with invalid assumptions and the type of barrier to which the assumptions led. (Since we only show edges greater than 10% and we exclude *insurmountable* barriers, the outgoing edges' of each node do not add up to 100%).

Selection barriers tended to lead to use barriers, suggesting that preventing invalid assumptions about programming interfaces' *capabilities* might avoid assumptions about their *use*. Furthermore, selection, coordination and use barriers—half of all observed—often led to understanding and information barriers. This implies that while debugging is a problem, a bigger concern is how prone VB's programming interfaces are to invalid assumptions *prior* to their use.

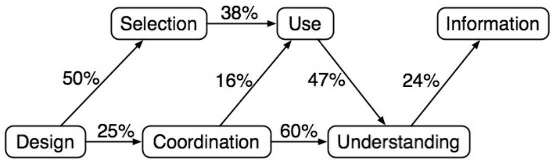

Figure 3. For *surmountable* barriers, the percent of each type overcome with invalid assumptions, and the type of barrier to which the assumptions led.

5.3 Are The Barriers Relevant to Experts?

Because experts are defined by their programming system experience, they tend to easily overcome selection, coordination, and use barriers. However, they do face design barriers because of the difficulty of their problems. This reflects the increasing importance of software architecture. Also, because professional programming systems often lack effective debugging tools [9], experts face substantial understanding and information barriers for which they must try to develop workarounds. This is evident in the use of print statements as all-purpose debugging tools.

5.4 Were There Data or Debugging Barriers?

Visualization tools assume difficulties in imagining the structure of data [14]. In our observations, learners did not face barriers in understanding data itself, but in trying to *act* on data (such as how to create or modify it). Because acting on data always involved a programming interface, such as a numerical operator or declaration, these barriers were categorized as selection, coordination, and use barriers. We do not know if more complex tasks would reveal barriers in understanding data.

"Debugging barriers" involved evaluating a program's external behavior relative to expectations (understanding barriers) and inspecting its internal behavior (information barriers). Because each task required different skills and tools to overcome, it was natural to distinguish the two.

5.5 Are The Barriers General?

We have used the six barriers to classify observations from studies of several end-user programming systems. For example, one common information barrier in the Alice programming system [3] is that variables' values are inaccessible at runtime because the output window is modal. This is in contrast to VB's information barriers, where the most common information barrier is determining where to insert print statements. This also in contrast to our ongoing observations of Macromedia Flash, where the most common information barrier is finding a particular line of code among hundreds of frames. Thus, in our experiences, the barriers are general enough to capture subtle but profound differences between the learning barriers of at least three diverse programming systems.

5.6 Are They Fundamental?

Because we have not applied the six barriers extensively, we cannot claim that these six describe *all possible* categories of learning barriers. With more data and more experience with the barriers, other categories may be apparent. For now, it seems more important to consider how well the six barriers support the design of more learnable end-user programming systems with respect to learning. In particular, do they help:

1. Evaluate existing programming systems?
2. Explore new areas in the design space?
3. Evaluate specific design choices?

The barriers have great potential in this regard: they identify important issues for existing systems, while providing usability guidelines for future systems.

6. A Learner-Centric Design Metaphor

Although the barriers seem helpful, we face the same challenge in using them for design as with any list of categories. For instance, one way to use them is to derive six design heuristics:

- Provide solutions to a domain's difficult problems.
- Offer facilities for finding programming interfaces that achieve particular behaviors.
- Show how to coordinate programming interfaces to achieve common behaviors.
- Programming interfaces should suggest for what they can be used and how to use them.
- Reveal what a program did or did not do.
- Help inspect a program's internal behavior.

However, these appear too vague to be helpful. We propose a *design metaphor* as an alternative. Metaphors are powerful because they can account for well-known properties of a design space, while supporting concrete analogical reasoning from a source to target domain. Ideal metaphors for end-user programming systems would (1) have a rich, human-centric source domain, (2) account for the six learning barriers, (3) be abstract and computer-centric enough to describe a variety of programming systems, and (4) be concrete enough to support analogical reasoning.

In Figure 4 we plot several existing computational metaphors against concreteness and human-centricity. Most are unsatisfactory; for example, "programs are objects" is too abstract to support rich analogical reasoning. Metaphors from end-user programming systems such as spreadsheets are too concrete to describe a variety of programming systems. Worse yet, all of the metaphors in Figure 4 lack even the slightest mention of human involvement.

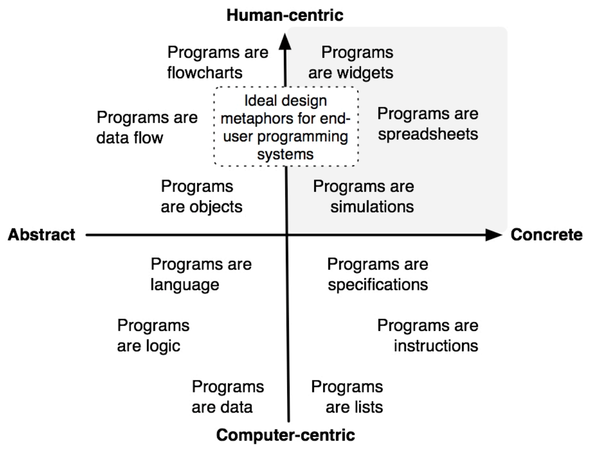

Figure 4. Ideal metaphors (the dashed box) would be abstract enough to describe many systems, but concrete enough for analogical reasoning.

6.1 The Factory Metaphor

Here we describe a metaphor that seems to satisfy our criteria. As a reminder, this is intended to help *programming system designers* think about learning barriers, and is not necessarily intended for end users.

A program is a *factory* and the learner is a factory's creator. Factories are made of *machines* (programming interfaces), which are coordinated to systematically receive, manipulate, and produce *products* (program output and behavior). In general, learners have many tools to help create, run, and inspect their factory (the programming environment).

Design barriers. Some products are difficult to produce, so designing "experimentally" can lead to unreliable factories (spaghetti code). Thus, it is often better to design a factory before building it (software engineering), but it is even easier to copy an existing factory that has been designed for such products.

Selection barriers. There are many different *machine suppliers* (programming systems) that may be used to design a factory, each offering a basic set of general purpose machines (language constructs) and a larger set of special-purpose machines (libraries). Machines may have a variety of levers, switches, and buttons (parameters), or they may have no controls at all (i.e., no-op). Some suppliers' machines have nearly identical affordances (i.e., Lisp syntax), while others' have entirely different affordances, but nearly identical behavior (i.e., *switch* statements vs. an *if-else-if* blocks). Suppliers offer catalogs to help learners select and use machines (help), but they tend to only explain what machines do, rather than how they might be used.

Use barriers. Because each machine has unique semantics, constraints, and feedback, it is difficult to know for what each can be used, how to use them, and what behaviors to expect. This is complicated by the diversity of machines: some pass products to other machines (procedure calls), some repeat a process (loops), and some transform a set of products into new products (expressions). Worse, most machines must be assembled (a character at a time), and how a machine fits together has little to do with what it does (syntax). Some suppliers offer manuals (documentation), but they seldom describe how to assemble machines.

Coordination barriers. In addition to choosing and using machines, learners must coordinate them in purposeful ways. Because it is not always clear which machines can fit together (syntax) and how they might interact (semantics), learners may visit other factories to find groups of machines that perform the desired behavior (example code). Learners then replicate the design and reconfigure the machines to match their own needs. Learners also consult manuals for hints on how to coordinate various machines.

Understanding barriers. Learners check their progress by running their factory before it is complete (testing). Sometimes a factory does not start because a machine is misassembled (syntax error): the challenge is to find the misassembled machine and find out what was right and wrong. Even if the factory starts, machines may fail. Some screech loudly when failing (exceptions); others fail silently, causing problems down the line. As a result, it is often difficult to know what a factory did or did not do, let alone why.

Information barriers. Once learners have an idea of what went wrong, they try to confirm it by making measurements and observations (debugging). This involves inspecting machines' interactions and probing those that are not open for inspection (code from compiled libraries). Some machines' behavior can be seen from a distance (e.g., which branch of a conditional is taken), but others require close inspection (e.g., an expression's dataflow). Suppliers offer tools to run factories step-by-step (debuggers), but most learners find it easier to install probes in just the right places (print statements). Finding a particular machine can be tricky (code navigation), but finding a faulty product is often much trickier (dataflow).

Variations in products. Some machines require products to have IDs (names). Although most machines are designed to manipulate only nearby products (static scope), many can retrieve products remotely with a product's ID (referencing). Some machines are picky about a product's type (strong typing); others can manipulate a variety of types in similar ways (polymorphism). Many suppliers offer machines that store and manipulate complex products such as documents, lists, images, and databases (abstract data types). These complex products are made out of simpler products (primitive data types).

Most suppliers make a sharp distinction between products and machines, though some offer machines that *contain* products (objects). Others offer machines that are products themselves (lambda functions). Machines handle products in a variety of ways: some pass products from machine to machine (dataflow), some take turns manipulating a sizeable collection of products (imperative), whereas some ship and receive products to and from the outside world (input/output).

Variations in control. Suppliers offer different ways of coordinating machines. Some require machines to run one at a time (single-threaded); others allow machines to run concurrently (multi-threaded). Some machines pass and return products to each other (functional); others broadcast and respond to events from in and outside of the factory (event-based). Some machines monitor the factory, ensuring compliance to rules (rule- and constraint-based). Others manage the flow of products in and out of factories (dataflow).

6.2 Using the Factory Metaphor for Design

If we regard the factory metaphor as a reasonably valid interpretation of modern computation, there are many ways that it can be used to support design.

For example, because factories, machines, products, and suppliers offer such rich source domains, the factory metaphor is an ideal foundation for analogical reasoning about the design space of end-user programming systems. In Table 2, we describe several criticisms of modern programming systems in terms of the factory metaphor, and offer design suggestions analogically derived from the source domain (i.e., catalogs, machine assembly, learning kits, and controls). As a result, many observations are far more concrete than anything that could have been derived just from design heuristics.

Another way to use the factory metaphor is to identify places where it breaks down. This reveals significant conceptual differences between the computational and physical worlds:

- Factories operate at a factor of the speed of light.
- Products can be made at runtime, requiring learners to reason about them before they exist.
- A product's type severely limits what it can be used for and what can use it.
- Machines can affect products remotely with IDs.
- Machines can themselves be used as products.
- Factories are made of many smaller factories and may be part of much larger factories.

Many of these properties have been shown to be difficult for learners to comprehend [5].

6.3 Actualizing the Metaphor

There are several systems that actualize interpretations of the factory metaphor. For example, in Tanimoto's *Data Factory* [16], data flows along conveyor belts, and is manipulated by machines along the way. In *The Incredible Machine* [6], players place physical artifacts in order to influence physical objects'

Design Barriers	Possible Solution for Programming System Designers
• Learners must recreate factories for difficult-to-produce products.	Offer these factories pre-built to learners, so that they do not have to solve hard problems. To find out which factories to provide, track the types of factories that are commonly built using the machines.
Selection Barriers	
• Some suppliers don't offer catalogs. If they do, the catalogs rarely say for what machine can be used.	Offer catalogs that are easy to flip through and that can be bookmarked. It should be easy to compare machines' roles in achieving particular behaviors. Catalogs should suggest for what types of behavior a machine might be used, what other machines might be required, and how to coordinate them.
• Some suppliers offer too many types of machines with similar functionality.	Standardize and simplify machines, only including machines that your target market would want to use. Track which machines are used and stop offering unused machines. Improving the usability of commonly used machines.
Coordination Barriers	
• Fitting machines together is difficult, unpredictable and error-prone.	Design machines to have simple and standardized interconnects, so that it is obvious whether two machines may be connected. Give feedback when learners are about to put machines in invalid places and explain the problem.
• It is difficult to imagine all of the ways that machines might be coordinated.	Provide example schematics to learners, so that they may know how machines might be coordinated. Look at the ways learners have been using your machines to get a sense for which schematics to provide. In addition to schematics, you might provide starter kits that learners can use directly in their factories.
• It is difficult to know how machines will interact when running.	Design machines so that they may be tested stand-alone or in small groups. They might even give some simple indicators of what they will do when turned on (such as indicating to which machine it will pass a product).
Use Barriers	
• Because machines require assembly, it is hard to tell where one machine ends and another begins. Also, machines don't always stay together when moved.	If learners *must* assemble the machines, at the very least provide good diagrams and mechanisms for doing so. It would be more ideal if machines came preassembled and encased, just like any other consumer product. Learners could simply take machines off a truck and set it down. It would be easier to place feedback on the exterior of the machine, because there would be less clutter on the outside of it, and machines wouldn't fall apart when moved.
• Most machines look the same and have similar interfaces. As a result, they tend to have very poor usability.	Carefully design machines' constraints and feedback to suggest their available actions, how to perform them, and what their effect will be. Use appropriate controls for each parameter (e.g., if a machine has a yes/no parameter, make it a switch instead of a dial). Arrange the controls to convey internal semantics of the machine.
Understanding Barriers	
• Learners can only see what goes in and out of a factory while it's running. With work, the factory can be run in steps.	Allow learners to watch their entire factory run. It would be easier to know what a factory did or didn't do if learners could walk from machine to machine and watch them operate, opening up individual machines for inspection. Provide ambient cues about what is happening while a factory is running (as when machines grind when failing).
Information Barriers	
• Finding a broken machine or products is difficult and time consuming.	Suppliers should offer facilities that inventory a learner's machines and the products that are generated and used, and provide interfaces to help a learner quickly and easily search the inventory to locate a machine or product.
• Inspecting relationships between machines and products is difficult.	Machines should record with which machines they interact and what data they use or create, so that learners can ask questions such as "*Where did this product come from?*" or "*which machine ran before this one?*"

Table 2. Barriers in today's programming systems and possible solutions, in terms of the factory metaphor.

trajectories. In *ToonTalk* [6], learners directly manipulate computational objects to create animations. These are examples of systems that avoid significant barriers while preserving computational features.

Of course, programming systems that do not actualize the factory metaphor can also benefit from a careful consideration of the factory metaphor. For example, by mapping the properties and concepts in VB to factory metaphor equivalents, we can apply all of the suggestions in Table 2.

7. Conclusions

With a greater understanding of the learning barriers in programming systems, we have a clearer notion of the central challenges in designing more learnable end-user programming systems:

- *Design is inherently difficult.* To overcome design barriers, learners need creativity. Programming systems should help scaffold creativity with salient examples and other forms of inspiration [13].
- *Finding behaviors is difficult.* To overcome selection barriers, learners need help searching for behaviors. Also, as more behaviors are offered, current tools will be increasingly ineffective.
- *Invisible rules are difficult to show.* To overcome coordination barriers, learners must know a programming system's invisible rules. Today's systems lack explicit support for revealing such rules, merely implying them in error messages.
- *Textual programming interfaces are limited.* To avoid use barriers, the feedback and interactive constraints of *every* programming interface must be carefully designed to match its semantics. The textual, syntactic representations of today's systems make this goal difficult to achieve.
- *Behavior is difficult to explain.* Overcoming understanding and information barriers requires some explanation of what a program did or did not do. We have made progress with the Whyline [7], but significant challenges remain.

We believe that these challenges are tractable, and hope that the six learning barriers and factory metaphor will help designers explore new ideas more efficiently and with more success.

8. Acknowledgements

We thank the students of Programming Usable Interfaces for sharing. This work was partially supported by NSF grant IIS-0329090 and the EUSES Consortium via NSF grant ITR-0325273. Opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect those of the NSF.

9. References

[1] "Occupational Outlook Handbook," U.S. Dept. of Labor, Bureau of Labor Statistics 2004, http://stats.bls.gov/oco.

[2] A. Blackwell, "First Steps in Programming: A Rationale for Attention Investment Models," IEEE Symposia on Human-Centric Computing Languages and Environments, Arlington, VA, Sept. 3-6, 2002, 2-10.

[3] W. Dann, S. Cooper, and R. Pausch, *Learning to Program with Alice*: Prentice-Hall, 2003.

[4] A. Engebretson and S. Wiedenbeck, "Novice Comprehension of Programs Using Task-Specific and Non-Task-Specific Constructs," IEEE Symposia on Human-Centric Computing Languages and Environments, Arlington, VA, Sept 3-6, 2002, 11- 18.

[5] J.-M. Hoc and A. Nguyen-Xuan, "Language Semantics, Mental Models and Analogy," in *Psychology of Programming*, *Computers and People Series*, J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore, Eds. London: Academic Press, 1990, 139-156.

[6] K. Kahn, "Drawings on Napkins, Video-Game Animation, and Other Ways to Program Computers," *Communications of the ACM*, 39(8), 1996, 49-59.

[7] A. J. Ko and B. A. Myers, "Designing the Whyline: A Debugging Interface for Asking Questions About Program Behavior," CHI 2004, Vienna, Austria, April 24-29, 2004, 151-158.

[8] A. J. Ko and B. A. Myers, "A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems," Submitted for publication.

[9] H. Lieberman, "The Debugging Scandal and What to Do About It," *Communications of the ACM*, 40(4), 1997, 26-78.

[10] D. A. Norman, *The Design of Everyday Things*. New York, NY: Doubleday, 1988.

[11] J. F. Pane and B. A. Myers, "Usability Issues in the Design of Novice Programming Systems," Carnegie Mellon University, Pittsburgh, PA CMU-CS-96-132, August 1996, http://www.cs.cmu.edu/~pane/cmu-cs-96-132.html.

[12] J. F. Pane, C. A. Ratanamahatana, and B. A. Myers, "Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems," *International Journal of Human-Computer Studies*, 54(2), 2001, 237-264.

[13] C. Quintana, J. Krajcik, and E. Soloway, "A Case Study to Distill Structural Scaffolding Guidelines for Scaffolded Software Environments," CHI 2004, Minneapolis, MN, 2002, 81-88.

[14] P. Romero, R. Cox, B. d. Boulay, and R. Lutz, "A Survey of External Representations Employed in Object-Oriented Programming Environments," *Journal of Visual Languages and Computing*, 14(5), 2003, 387-419.

[15] J. G. Spohrer and E. Soloway, "Analyzing the High Frequency Bugs in Novice Programs," Empirical Studies of Programmers, 1st Workshop, Washington, DC, June 5-6, 1986, 230-251.

[16] S. Tanimoto, "Programming in a Data Factory," Human-Centric Computing Languages and Environments, Auckland, New Zealand, Oct. 28-31, 2003, 100-107.