How to Teach a Fish to Swim

Stephen Farrell Paul P. Maglio Christopher S. Campbell
IBM Almaden Research Center
650 Harry Rd.
San Jose, California 95120
{*sfarrell, pmaglio, ccampbel*}*@almaden.ibm.com*

Abstract

We have developed a virtual fish tank in which computer users are represented by animated fish. The actions and interactions of the fish in the tank are meant to reflect the actions of users in the real world. Our first attempt at creating a programming environment that allowed people to customize their own fish did not work very well because users did not want to explicitly write programs to control their fish. Maintaining the fish tank metaphor, we attempted to solve this problem by having users teach *fish rather than write code. We borrowed ideas from the literature on programming by demonstration and developed a method of* programming by conditioning *in which users demonstrate behaviors and also reward (or feed) fish that are behaving appropriately. Rewards give users the ability to define high-level behaviors (sets of specific movements) and complex relationships between situations and responses.*

Figure 1. A user tapping on the large-screen fish tank display in our lab.

1. Introduction

The fish tank is a virtual space in which people and programs can be represented as animated fish. This space was initially conceived as a fun way of representing people involved in a synchronous chat session. Once implemented, however, the fish tank became a general framework for supporting group awareness and group interaction [1]. In general, fish in the tank can respond to external events (e.g., taps or mouse clicks on the screen) and to internal events (e.g., the presence or motion of other fish or objects in the tank), and can also generate new events (e.g., moving from one point to another, or delivering a text message to the tank). We have observed that the tank is especially compelling when displayed on a large screen with a touch-sensitive surface (see Figure 1).

A fish can be thought of as a social proxy [2]. That is, an individual fish can stand in for an individual person when interacting with others who are "in the tank" at the same time. Such fish are social proxies to the extent that their behavior in the tank conveys aspects of their behavior in the real world. For example, when a group of people are actively engaged in a conversation, their fish might school together, following each other around in the tank. But if a particular person is not actively involved in chatting or otherwise interacting with others, this person's fish might swim away from other fish in the tank. The mere appearance of a fish in the tank might signify a person's availability. In these and other ways, the fish, their appearance, and their behavior can all be used represent and affect what is actually happening among a group of people.

We have tried to keep all aspects of the tank within the metaphor of fish swimming around in a tank (cf. [3]). For instance, people interact with the system by tapping on the screen, and people are represented by what look like fish, and the fish move or swim in distinctly fish-like ways—there are no menus, scrollbars, or windows. Of course, we have had to take some liberties with the metaphor; for instance, our fish can talk, which is conveyed by little bubbles of text that float upward from their mouths. In any event,

we believe there are many benefits to sticking close to the fish tank metaphor, including simplicity of interaction and leveraging people's ability to gain useful, high level information about the state of the group by simply glancing at the tank. Perhaps more importantly, we believed that the fish metaphor would enable people to be creative in determining how their fish look, move, and behave.

1.1. Fish Programming is Hard

We created a Java-based *Fish Development Kit* for people to implement and program their own fish. This Fish DK allowed programmers to specialize a `Fish` class to control a fish's appearance and behavior. To do this, a fish developer would write code to handle various events such as taps on the screen, to navigate the tank with the help of a library of geometry functions, and to interact with other fish with the help of an internal fish position and velocity model. Programmers were left on their own to figure out how to determine a person's state in the world (e.g., based on email, motion sensors, telephones, etc), and how to represent that state through fish appearance and behavior.

Our hope was that the fish tank would provide a space in which people represented themselves creatively and fostered group awareness and interaction. The success of the tank depends on group members' abilities to program their own fish. Yet even for our computer science research group, fish programming turned out to be far more difficult than we had anticipated. Though a couple of us had great fun writing fish, few people overall found the energy and time to write the code to respond to fish tank events and to move fish around the tank. We were left wondering how to make fish programming easier.

In what follows, we describe our new approach to fish programming, first in general terms and then more formally. We have adapted programming by demonstration to the fish tank, adding the notion of rewards to reinforce behavior. Our contribution lies in the application of the general end-user programming approach to this domain, as well as in our novel use of rewards. In addition, we discuss our end-user programming approach in the context of developing fish that act as social proxies, conveying formation about the state of the user by their appearance and behavior in the tank.

2. Fish Programming by Conditioning

Our primary goal is to enable people to create fish that can act as social proxies, conveying something about a person's state through appearance and behavior, and supporting natural interactions with others in the tank. Following the fish tank metaphor, we began to conceive of "programming a fish" as "teaching a fish". The experience of training a fish to behave ought to be somewhat like the experience of training a real animal or pet, requiring both demonstrations and rewards. One might want to teach a fish to follow another fish, to swim "aggressively," to swim in an "SOS" pattern, and so on. In these cases, it might make sense to show the fish how to move, and then to feed the fish when it behaves appropriately. Because our new fish programming approach combines demonstration with rewards much as the operant conditioning of behaviorist psychology, we call our approach *programming by conditioning* (PBC).

Like training an animal, programming is an attempt to transfer a model of activity from one system (e.g., the head of the trainer or programmer) to another (e.g., a dog or a computer system). Unlike training an animal, programming by writing code requires using a definite language that both programmer and computer understand, which means that programmer and computer system need not assume or infer anything about what was meant by some specific term or action. Like training an animal, programming by demonstration requires the computer system to infer what was meant by the specific actions taken, as the shared language leaves much unstated. In the case of training a fish in our tank, for example, when demonstrating a behavior such as moving to the left, the fish must not only determine the direction, left, but also the conditions under which to move left, such as when it reaches a certain horizontal position in the tank or when there is another fish immediately to the right (cf. [4]). Considered this way, teaching a fish by demonstration runs into all the familiar programming-by-demonstration problems of generalizing from examples (e.g., [5]). By adding rewards to programming by demonstration, we believe we have a novel way of facilitating generalization.

2.1. Why Programming by Conditioning?

Our PBC approach improves programming over our original Fish DK along two dimensions: abstraction and programmer experience (see Figure 2.1). First, the abstraction level moves from concerns of absolute positioning to those of relative positioning and then to aggregation of responses into large-scale behaviors. Second, the programmer experience changes from writing code to direct manipulation of the fish and then finally to training. We now consider these two dimensions in more detail.

Shown along the vertical axis in Figure 2.1, "programmer experience" refers to the way in which the users express how they want their fish to behave. We chose this term by analogy to "user experience," as users and programmers are the same in this context. At one extreme is good old-fashioned coding, which involves learning a programming language, development tools, and the fish application programming interface (API). In the middle is programming by demonstration, which for the fish tank might involve di-

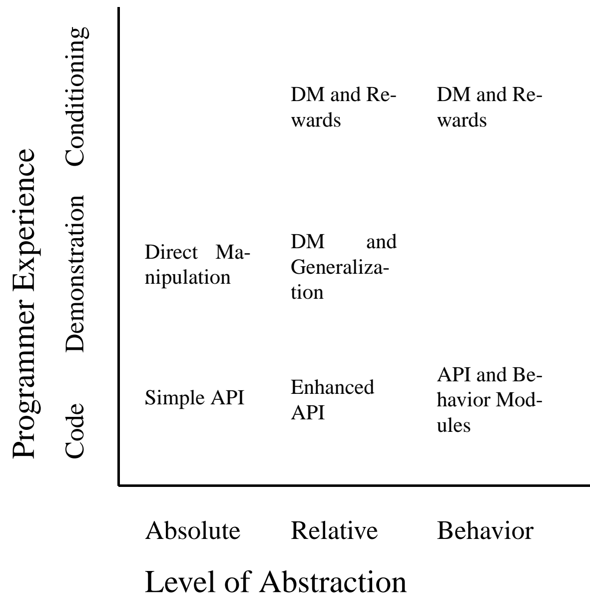

Figure 2. How to improve the fish programming experience.

rectly manipulating the fish to indicate actions. The system would use these examples to create general rules to specify how the fish acts in the tank. Correcting or reinforcing behavior might be done by providing additional examples. Finally, at the other extreme is PBC, which adds the notion of "reward" to programming by demonstration. This allows the user the opportunity to cement behaviors that the fish has correctly inferred and executed.

The horizontal axis, "level of abstraction," refers to the kinds of concepts programmers can express to the system. At one extreme, positions and actions in the tank are referred to in absolute terms (e.g., move from coordinate a, b to coordinate c, d at velocity v). This has the advantage of being succinct, but in most cases is not how a programmer would think about a fish's actions. The next level refers to motion relative to other objects, or to responses to events (e.g., move toward a wall, or move away from an approaching fish). This abstraction level might be sufficient for many tasks, but it restricts the user from referring to sequences of actions that together make up a large-scale behavior. In the fish tank, the highest level of abstraction refers to aggregations of actions or responses—speaking directly about large-scale behaviors (e.g., a user might want a fish to "be afraid" of another fish).

As mentioned, we are following the fish tank metaphor to discover whether it makes sense for users to program fish in the same way that they use the fish tank [6]. People program fish by directly moving them around the screen, providing an example of appropriate behavior. Of course, this strains the metaphor of the fish tank—but only a little. After all, people can train dogs to sit by pressing down on the dogs' backs. The relationship between the manipulation and what the fish should do is at the lowest level of abstraction: if the fish is in position a and the user drags the fish to position b at velocity v, then the next time the fish is in position a it should move to b at velocity v. Unfortunately, things become more complicated when attempting to generalize the conditions under which the fish should move this way, as well as what it means to move "this way". The user might want the fish to move away from another fish, or the user might want the fish to move to a specific spot. It is not easy to see how the user can express the former by directly moving the fish. As a result, programming by example must rely on the system to make inferences. Our third level of abstraction, the ability to characterize and refer to sets of responses or actions, seems unattainable using only the constraining mechanism of dragging. This is why we introduced rewards.

The reward mechanism enables the user to interact with the fish at a higher level of abstraction because the rewards refer generally to how the fish has been behaving rather than specifically to a certain action. For example, if a user wants a fish to be "shy" he or she can reward it after several events in which the fish swims slowly and swims away from other fish, thus associating these two responses and making it more likely to correlate these actions in the future. One drawback of the conditioning approach is the reliance on serendipity—that the user will observe fish performing a sequence of actions that ought to be reinforced.

3. New Fish Programming Model

The user provides a fish with examples of actions and with positive rewards for sequences of actions. By paying attention to these data and when they are given, a fish ought to be able to (1) generate new actions that are *like* those given in the examples, and (2) combine actions in a way that is *like* the action sequences that have been rewarded.

3.1. Give a Fish an Example

People who used our original fish tank often wanted to drag their fish around with the mouse (or equivalently, with their fingers when using a touch screen). This intuitive action seemed the obvious sort of thing to leverage when adding end-user programming support. Thus, we allow the user to watch the activity in the tank and, at any time, drag a fish to where he or she wants it to go. For instance, the user might drag it toward another fish if the user wants one fish to follow the other.

More precisely, the example provided by the user is interpreted as moving the fish from an initial to a final location in the time period between the mouse click and mouse release events. The fish records the `move` action and the initial `state`. In the case of `move`, the action is simply a velocity in a particular direction. The `state` is a snapshot of all attributes the fish can pay attention to in the tank. A `state` is comprised of the relative motion of all other fish and the walls, as well as the current location and motion of itself. These are referred to as `attributes` of the `state`. Most attributes, such as velocity, take on numeric values. To facilitate generalization, extra attributes can be added; for example, there are attributes for the relative velocity of "the left wall" and "any wall". Likewise, the state could contain information about things outside of the tank, such as information about the user or about the weather. In addition to state attributes, there are fish tank `events`, such as when a user taps on the screen or when a fish changes direction. A history of these events are included in the `state` information, each associated with relevance scores that diminish over time.

A fish remembers each example it is given. Over time, a fish develops a library of examples. By continuously comparing the current state in the tank to this library, the fish can determine when and how to react. This library or `example-table` looks like this:

$$\begin{bmatrix} S_1 & \rightarrow & A_1 \\ S_2 & \rightarrow & A_2 \\ ... & & ... \end{bmatrix}$$

The `example-table` maps from `states` S to `actions` A. Both `state` and `action` are represented as tables of attribute-value pairs:

$$\begin{bmatrix} a_1 & = & v_1 \\ a_2 & = & v_2 \\ ... & & ... \end{bmatrix}$$

An `action` may include the values for the attributes x and y of the new direction/velocity vector. A `state` may contain many attributes including, for instance, the direction/velocity vector for every other fish in the tank.

3.2. How Fish Generalize

When not recording examples, a fish continuously samples the state of the tank. The fish needs to respond to situations that are *like* the initial states of the stored examples. One approach might be to simply compare all attributes of the current state to all attributes of the example states to find cases where all of the attributes match. This might work well precisely in those circumstances when the current state matches an example in the table, but this would happen rarely. And, more importantly, this is probably not what the user wanted to convey. For instance, the user might mean "when near the wall" rather than "when in position x, y going at velocity v with left wall at distance d and fish 1 in position a, b and ..."

Our fish generalization process works by grouping sets of examples around the attributes they have in common. For instance, all examples in which a fish is moving away from a particular approaching fish will be grouped together. We maintain the groupings of examples based on state attributes in the `generalization-table`:

$$\begin{bmatrix} a_1 & \rightarrow & G_1, G_2, ... \\ a_2 & \rightarrow & G_1', G_2', ... \\ ... & & ... \end{bmatrix}$$

where $a_1, ...$ represents all the attributes of a `state` and each G represents a `generalization`. A `generalization` is composed of a set of `examples` $[E, ...]$. The `examples` it contains are similar to each other with respect to the associated attribute. The meaning of the term, "similar" will depend on the particular implementation—for example, one could divide the range of possible values into 10 buckets and drop each example into the closest one. Each `example` occurs exactly once in each row.

When a fish examines the attributes of the current state of the tank, it tries to find appropriate generalizations (G_i) in the `generalization-table` for each attribute (a_j). There may or may not be a `generalization` corresponding to the current value of a particular state attribute. If any `generalizations` are found, then the current state resembles a set of example states with respect to one attribute. If many `generalizations` are found, then the current state resembles many sets of example states with respect to many attributes. If, for instance, several examples showed the fish moving away from a particular fish, and several others examples showed the fish moving away from the left wall, then the current state might be that the fish is near that particular fish *and* is near the left wall. In this case, two potentially appropriate generalizations have been found.

3.3. How Fish Decide What to Do

A fish must chose an action in response to its current circumstances. Recall that each `generalization` contains a set of `examples`, and that each example contains a `state` and an `action`. A fish ought to act *like* it has been taught to act—taking actions that are somehow based on the examples it has been given. In this context, we use a rather literal interpretation of *like*: We define an `option-range` as a set of attribute-value pairs in which the `attributes` are identical to those for an action, but in which the values

specify *ranges* of possible values that represent the variation of values in the examples given. For example, the option-ranges that correspond to the generalization of "being near the left wall" would allow for much leeway in moving up, moving down, and moving right, but would exclude moving left. An option-range can be thought of as the disjunction of the actions from each of the examples in a generalization.

Thus, for each generalization, a fish computes a range of options. If there is only one generalization, then a new action can be easily based on that range. In any event, to create a new action, the fish can simply choose random numbers for each attribute bound by the minimum and maximum of the range. To take into account more than two examples, the fish can use a probability distribution in which greater weight is given to the parts of the range that correspond to more examples it has seen.

3.4. How to Reconcile Ranges of Options

If more than one generalization contains an attribute that corresponds to the current state, these competing situations must reconciled by merging their option-ranges. A trivial solution would be to pick one at random and throw the others out. A more realistic solution would be based on the intersection of the option-ranges. For example, it might be easy to merge option-ranges when a fish is moving into a corner of the tank. In this case, the fish might be near both the left wall and the top wall. The option-range for the generalization of being near the left wall of the tank might be move up, move down, or move right. The option-range for the being near the top of the tank might be move left, move down, or move right. Taking the intersection of these, we see that the fish can move right or move down.

Reconciling generalizations is more difficult when the intersection of option-ranges yields no action. This might happen, for example, if one generalization says to flee from a fish, but another pushes the fish into a corner. In this case, one option-range might allow only moving left, and another only moving right. They are mutually exclusive. Of course, this is a *real* problem that could only be avoided by a little planning or preparation, but for simplicity our fish only react to events. In this case, a couple of heuristics might help: (1) favor the option-range with the least variation, (2) favor the option-range that prescribes the greatest velocity, (3) pick one at random.

3.5. How Fish Decide When to Act

Choosing *when* to act can be as difficult as choosing how to act in some programming-by-demonstration systems [4]. In our approach, however, it is trivial. After a fish computes an option-range, it checks to see whether the last action was contained in that range. If it is not, then the fish chooses and executes a new action. In other words, a fish only reacts when circumstances have changed.

3.6. Give a Fish a Reward

When a user observes a fish performing sequence of actions that he or she approves of, the user rewards the fish by feeding it. For example, if the fish flees from other fish a few times in a row, the user can drop some food next to the fish, encouraging the fish to be shy. In addition to reinforcing the action the fish had just taken, our reward model also reinforces the association of recent actions to increase the chance that they occur in sequence in the future. Rewards are how the user communicates approval of an aggregation of actions (also called a *behavior*).

Internally, rewards are assigned to generalizations in the form of incrementing a weight w—thus, generalizations really consist of a set of examples and a weight. The higher the weight, the more the option-range corresponding to that generalization will weigh in the action chosen. This weight value is particularly helpful in situations in which two option-ranges conflict with each other, as the one with the lower weight can be discarded.

Upon receiving a reward, a fish considers its most recent actions and increments the weight of the generalization on whose option-range that action was based. The assigned weight is inversely-proportional to the time between action and reward. So the weight is increased much more for an action that occurred one second prior to the reward than for an action that occurred 8 seconds prior. For instance, the time-weight curve probably ought to be exponential rather than linear, dropping to near zero after about 10 seconds.

Another possibility is that several generalizations contribute to the final option-range. This is easy to deal with: the weight can be incremented proportionally to the similarity between the option-range associated with each generalization and the final option-range.

In addition to evaluating generalizations relative to one another, rewards also increase the likelihood that generalizations will group together in a sequence. To achieve this, a list of generalizations and weight pairs (the generalization-history-list) is added to a generalization:

$$G : [E, ...], w, ([G', w'], ...)$$

where G' might be any other generalization, including G itself, however each G can only occur once in this list. In addition to incrementing the weight w of G, rewards

increment the value of w' for G' where G' is the generalization used in the previous action. If many generalizations were involved, this can be handled as described previously.

Consider the case in which a fish chooses a new action based on many generalizations $[G, ...]$. The fish looks up the previous generalization G' in the `generalization-history-lists` for each G. If it finds G' in this list, it increments the weight it is associating with G by an amount scaled inversely with the time since the previous action. This augmented weight is then used to favor that generalization over others when choosing an action.

The combination of examples and rewards allows the fish to associate chains of actions. These associations are established by rewarding a fish when it executes an appropriate sequence of steps. Such a sequence might correspond to a large-scale sophisticated behavior, such as chasing another fish, rather than to low-level action, such as moving left.

4. How Fish See People

We have focused on a model of how a fish can learn to behave when instructed by a person through examples and rewards. As mentioned, however, our larger goal is to make a fish's behavior reflect the state of a user. Doing so poses two problems: (1) how to find out about the state of the user, and (2) how to make the fish react to this state.

There are many ways to instrument an office to reveal information about a user's state. For example, one can look at incoming and outgoing email, to whom someone is talking on the phone, or even who else is in the office (e.g., [7]). More low-level information can be gathered with motion detectors and looking at the state of the screen saver or keyboard and mouse events (e.g., [8]).

Once attributes of a user's state are available electronically, there are several ways we might work them into the fish's model of the world. The simplest approach would be to think of a person as having a small number of detectable states he or she would like reflected in the tank, such as "at home", "available", "busy", "out-to-lunch", and so forth. The user might then teach the fish how to behave in each of these situations. For example, when the user's state is "at home," the user could direct the fish to swim off the screen. Internally, the fish would group all the examples it had learned by the state of the user, maintaining an `example-table` and a `generalization-table` corresponding to each state. This approach might work reasonably well, but does not seem particularly elegant. Furthermore, the fish would not be able to share behaviors it had learned from one state with those it learned in another.

A more interesting approach would be to incorporate aspects of the user's external world state into the internal `state` that the fish is aware of. Because our model avoids making any assumptions about the meaning of state attributes, fish can respond to things in the real world just as easily as they respond to state and events inside the tank. For example, a fish might learn to respond to a user entering the office (based on motion detectors in the office) just as it had learned to respond to being approached by another fish. Likewise, a user might drag a fish toward another user's fish immediately after sending that user email, offering the fish a reward after it follows the other user's fish for a bit. In this case, the user's fish could learn this "following" action and repeat it in the future, as the reward would have associated the `generalization` tied to the attribute of having sent mail to a particular user with the `generalization` tied to the action of following a particular fish.

5. Conclusion

Fish programming is done by demonstrating behaviors and rewarding appropriate actions. We have formalized our programming model by describing how to go from examples to generalizations, how generalizations figure in to action selection, and how rewards can be used to reinforce actions and associate actions with one another. Our new fish programming model starts from the assumptions (1) that fish users ought to be fish programmers, and (2) that fish ought to be programmed by dragging them around the tank and feeding them when they behave appropriately. We think this approach fits the fish tank metaphor fairly well—or at least better than explicitly writing code to control fish.

Instructible agents or systems learn through a conversational process between the user and the system in the form of examples, hints, and questions (e.g., [9, 10, 11]). The goal of instructible agents is to discover (a) what the user is doing and (b) when the user wants this action performed within a certain task model; for example, graphical editor layout [12], or text recognition [13]. The approach taken here differs significantly in that fish are trained as social proxies and do not perform detailed, direct control, or task dependent actions. The fish or social agent has the task of displaying the user's activities and personality to other users in the group. In other words, the fish is not an interface to perform explicit tasks, but rather a representation of the user to facilitate group member interactions. Since users are creating a representation of themselves, they are creating an agent that is like them generally, rather than agents for specific, special case jobs. Another benefit of the fish tank over instructible agents is that the metaphor allows for (1) the direct mapping of training animals to training fish and (2) a simple, pre-existing mental model of what fish know.

One drawback we noted with the rewards mechanism is its reliance on the user observing the fish performing the sequence of actions to be reinforced. Perhaps the user has in mind behaviors that he or she would like to reinforce, but the sequence only occurs infrequently. One way to allevi-

ate this potential problem would be to have the fish cycle through some different responses after an example, thus increasing the likelihood of the desired sequence appearing. Another approach would be to add negative as well as positive feedback, thus allowing the user to convey more information. By reducing the probability that fish will perform undesirable sequences, the fish is more likely to exhibit the sequences the user wants to reinforce. But all things being equal, we would prefer to avoid punishment.

Though we tend to think that users ought to be creative with the way they represent their own state in the tank, without a common visual language, it might be difficult for one user to understand what another is trying to convey. For example, one user's fish might swim slowly to indicate that the user is deep in thought and should not be disturbed, whereas another user might use slow movement to signal availability. One way to handle this would be to seed the environment by preloading the fish's `example-table`. In this case, fish would start with an off-the-shelf personality and users would be able to make minor adjustments to customize their fish. This approach would also save time for those too busy to tell their fish *everything*, and would avert the potentially confusing experience of having a fish that initially doesn't react to anything.

Although we have a reasonably well-defined plan of how our PBC fish tank should work, we have not yet completely implemented it. Doing so is clearly the next step. The code for the graphical user interface (the fish tank) is being extended to allow for dragging and rewarding interactions. Most of the work is in coding the fish to learn according to the design presented here. We are somewhat concerned about performance, as the sampling of the state would be frequent and their might be a lot of data, but there are many potential optimizations, such as caching static or infrequently changing values, and only factoring in the state of *nearby* fish, walls, and so forth.

In summary, we have developed a fish tank environment for creating social proxies, enabling awareness and interpersonal interaction in our work group. Because we found that traditional programming of the fish did not catch on with all members of our lab, we have adapted programming by demonstration to the fish tank setting. In the end, we hope the old saying is correct: "Give users a fish development kit and they'll write code for a day—let users teach their fish and they will train them for a lifetime."

References

[1] S. Farrell, "Social and informational proxies in a fish-tank," in *Extended abstracts of the Conference on Human Factors in Computing Systems (CHI 2001)*, New York, 2001, ACM Press.

[2] T. Erickson, D. N. Smith, W. N. Kellogg, M. Laff, J. T. Richards, and E. Bradner, "Socially translucent systems: social proxies, persistent conversation, and the design of "babble"," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI '99)*, New York, 1999, ACM Press.

[3] T. D. Erickson, "Working with interface metaphors," in *The art of human computer interface design*, B. Laurel, Ed. Addison-Wesley, 1990.

[4] D. Wolber and B. A. Myers, "Stimulus-response pbd: Demonstrating when as well as what," in *Your wish is my command: Programming by example*, H. Lieberman, Ed., pp. 321–344. Morgan Kaufmann, 2001.

[5] H. Lieberman, *Your wish is my command: Programming by example*, Morgan Kaufmann, 2001.

[6] B. Nardi, *A small matter of programming: Perspectives on end user computing*, MIT Press, Cambridge, MA, 1993.

[7] H. Yan and T. Selker, "Context-aware office assistant," in *Proceedings of the Conference on Intelligent User Interfaces*. ACM Press, New York, 2000.

[8] P. P. Maglio, R. Barrett, C. S. Campbell, and T. Selker, "An architecture for developing attentive information systems," *Knowledge-Based Systems*, vol. 14, pp. 105–112, 2001.

[9] H. Lieberman and D. Maulsby, "Instructible agents: Software that just keeps getting better," *IBM Systems Journal*, vol. 35, pp. 539–556, 1996.

[10] P. Maes, "Agents that reduce work and information overload," *Communications of the ACM*, vol. 37, no. 7, pp. 31–40, 1994.

[11] D. C. Smith, A. Cypher, and J. Spohrer, "Kidsim: Programming agents without a programming language," *Communications of the ACM*, vol. 37, no. 7, pp. 54–67, 1994.

[12] H. Lieberman, "Mondrian: A teachable graphical editor," in *Watch what I do: Programming by demonstration*, A. Cypher, Ed. MIT Press, Cambridge: MA, 1993.

[13] H. Lieberman, B. A. Nardi, and D. J. Wright, "Training agents to recognize text by example," in *Your wish is my command: Programming by example*, H. Lieberman, Ed., pp. 227–244. Morgan Kaufmann, 2001.