

ALEXANDER REPENNING, ANDRI IOANNIDOU

WHAT MAKES END-USER DEVELOPMENT TICK?

13 DESIGN GUIDELINES

ABSTRACT:

End-user development has enormous potential to make computers more useful in a large variety of contexts by providing people without any formal programming training increased control over information processing tasks. This variety of contexts poses a challenge to end-user development system designers. No individual system can hope to address all of these challenges. The field of end-user development is likely to produce a plethora of systems fitting specific needs of computer end-users. The goal of this chapter is not to advocate a kind of universal end-user development system, but to cut across a variety of application domains based on our experience with the AgentSheets end-user simulation-authoring tool. We have pioneered a number of programming paradigms, experienced a slew of challenges originating in different user communities, and evolved end-user development mechanisms over several years. In this chapter we present design guidelines that cut across this vast design space by conceptualizing the process of end-user development as a learning experience. Fundamentally, we claim that every end-user development system should attempt to keep the learning challenges in proportion to the skills end-users have. By adopting this perspective, end-user development can actively scaffold a process during which end-users pick up new end-user development tools and gradually learn about new functionality. We structure these design guidelines in accordance to their syntactic, semantic and pragmatic nature of support offered to end-users.

Keywords: End-user programming, agents, visual programming, programming by example, graphical rewrite rules,

1. INTRODUCTION

The fundamental aim of End-User Development (EUD) (Klann, 2003; Paternò, 2003) is to empower users to gain more control over their computers by engaging in a development process. The users we have in mind, called end-users, are typically not professional software developers. End-users employ pre-existing computer applications to achieve a variety of goals. They may be using email and browser applications to communicate with other users, word processors to write books, graphic applications to create computer art. Often, to make these applications truly useful, end-users may have to adapt these applications to their specific needs. Adaptation may assume many forms ranging from simple forms such as changing preference settings of applications, to more complex such as writing filtering rules for email applications or defining formulas for spreadsheets. The need to enable these more complex forms of adaptation is quickly increasing for various reasons. For instance, browsers are used to access quickly growing information spaces. Only the end-users of an application, not the developers of that application, can decide on how to deal with all this information. Application developers can no longer anticipate all the needs of end-users. This discrepancy between what application developers can build and what individual end-users really need can be addressed with End-User Development.

The term End-User Development is relatively new, but it stems from the field of End-User Programming (Bell & Lewis, 1993; Cypher, 1993; Eisenberg & Fischer, 1994; Fischer & Girgenson, 1990; Ioannidou & Repenning, 1999; Jones, 1995; Lieberman, 2001; Nardi, 1993; Pane & Myers, 1996; Rader, Cherry, Brand, Repenning, & Lewis, 1998; Alexander Repenning & Sumner, 1995). The shift from “programming” to “development” reflects the emerging awareness that, while the process of adapting a computer to the needs of a user may include some form of programming, it certainly is not limited to it. In that sense, most of the research questions from end-user programming carry over to end-user development but because of the widened scope of end-user development new issues need to be explored. End-User Development is of relevance to potentially large segment of the population including most end-users of traditional computer applications but also of information technology associated with ubiquitous computing. How, then, can the emerging field of end-user development provide answers to adaptation challenges including this wide range of applications, devices, contexts and user needs? How can we conceptualize this end-user and how can we help to make the process of end-user development as simple as possible?

Focusing initially on the programming aspect of end-user development we can benefit from research areas exploring strategies to make programming simpler. Visual Programming, for instance, has explored the consequences of replacing traditional, text-based, representations of programs with more visually oriented forms of representations. An early period of superlativism ascribing near magical powers to visual programming tried to polarize visual and textual programming approaches into good and bad. Many instances were found when textual programming worked just as well if not better than visual programming (Blackwell, 1996). Gradually, it was recognized that the question of visual versus textual

programming approaches cannot be decided on an class level but needs to be explored at the level of instances and closely investigated in the context of actual users and real problems. A number of frameworks have been postulated to evaluate programming approaches at a much finer level of grain. The Cognitive Dimensions framework by Green (Green, 1989; Green & Petre, 1996) introduced 14 cognitive dimensions to compare programming environments. Over time this useful framework has been extended with additional dimensions and a number of cases studies evaluating and comparing exiting programming environments.

A framework in support of evaluation does not necessarily support the design and implementation of systems. For this article we like to assume a more prescriptive position by providing a collection of design guidelines that we collected over a period of twelve years of developing, using and improving the AgentSheets simulation authoring tool. The majority of these guidelines emerged from user feedback initially from the AgentSheets research prototype and later the commercial product. Design intuition may initially be the only clue on building a system, but it will have to be replaced with real user experiences to be useful.

The process of end-user development is about learning. Many different users ranging from elementary school kids to NASA scientist have used AgentSheets over the years. Trying to reflect and generalizing over user populations and problem domains we found one perspective of experience that all of these users had in common. End-user programming, or for that matter end-user development, can be conceptualized as a learning experience. The process of end-user development is not a trivial one. End-user development environments cannot turn the intrinsically complex process of design into a simple one by employing clever interfaces no matter how intuitive they claim to be. Design cannot be addressed with walkup-and-use interfaces (Lewis, Polson, Wharton, & Rieman, 1990; Lewis & Rieman, 1993). We found the learning perspective useful because it allowed us to characterize the end-user as a learner and to create end-user development tools in support of learning.

The essence of End-User Development is, we claim, to scaffold a programming or development tasks as a learning experience. We can neither make any assumptions on what the problem that a user tries to solve is nor the usage context. However, we can make some assumptions about the motivation and background of an end-user. Similar to the person trying to program a VCR, an end-user developer is not intrinsically motivated to learn about programming or development processes. Programming is simply a means to an end. The goal is to record the show, not to create a program. The VCR programming task is not likely to be enjoyed. At the same time an end-user programmer is not likely to have a computer science background and also not typically paid to do end-user programming.

The appearance of an End-User Development system is largely irrelevant: this is not a question of visual versus textual programming. What is extremely important is that the End-User Development system carefully

- balances the user's skill and the challenges posed by a development process; and
- enables an end-user developer to gradually acquire necessary skills for tackling development challenges.

In short, what is needed is to conceptualize the process of end-user development as a learning experience.

1.1 Flow

A framework that has allowed us to explore design options comes from psychology. The notion of flow has been introduced by Csikszentmihalyi to analyze motivational factors in learning (Csikszentmihalyi, 1990). In a nutshell, the idea of flow is that optimal learning can take place when there is a proportional relationship between the challenges a task poses and the skills the learner has. Anxiety results if the challenges outweigh the skill, while boredom results if skills outweigh the challenges (see flow diagram in Figure 1). Assume a really experienced tennis player is matched up against a beginning player. The experienced player exhibits a large amount of skills. Playing against the beginning player will pose little of a challenge. The beginning player, in contrast, has almost no skills but will certainly perceive playing against the experienced player to be a high challenge. Putting these values into the diagram we see that the experienced player is likely to get bored whereas the beginning player is likely to enter a state of anxiety.

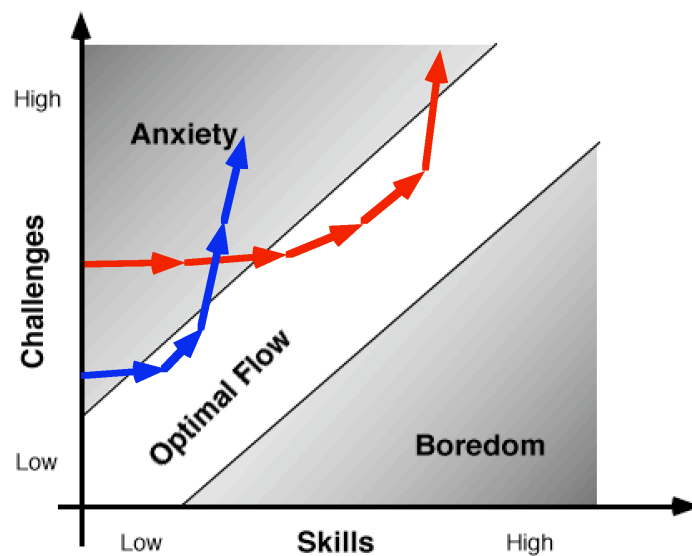


Figure 1. *Flow: The zone of optimal flow provides good learning conditions by balancing challenges posed to users with their skills. End-user programming requires low thresholds but may not necessarily scale well. Professional programming is less concerned with initial learning.*

We have used the notion of flow during several workshops on End-User Programming to discuss how users gradually learn different End-User Programming tools. In the computer context we look at the programming/development skills a user is likely to have and compare that to the perceived challenge of using an end-user development system. According to Csikszentmihalyi's theory the ideal learning situation is established by keeping the ratio of challenge to skill in the diagonal band of the diagram called optimal flow.

In the Syntactic, Semantic, and Pragmatic Guidelines sections of this article we will discuss design guidelines with respect to flow. A specific end-user development activity can be conceptualized as a single point reflecting development experience and problem complexity in the flow diagram. More importantly, repeated or long-term use of an end-user development system is captured as arrows indicating a transition in skills and challenge. This transition may be due to using the same end-user development system over time or could also be the result of transfer of using other systems.

The shape of skill/challenge transition chains reveals the usability profile of a system. A programming environment for professional programmers is significantly different from an end-user development system. The complexity of a professional programming environment such as Visual Studio is overwhelming even to new professional programmers. Because they are paid, however, they are likely to be willing to make a transition through a non-optimal territory including anxiety. End-user development systems typically cannot afford this without frustrating users. A simple development task for end-users needs to be sufficiently supported that even low skills will be sufficient to solve simple challenges without the overhead of a complete computer science education first. In most cases anxiety translates into giving up. Ideally, end-user development tools would strive for the goal of a “low-threshold, no ceiling” tool (Papert, 1980). Realistically, a low threshold will probably need to be traded for scalability. This may be acceptable, since nobody expects end-user programming environments such as a VCR programming interface to be scalable to the point where an operating system could be implemented with it.

The majority of our discussion relating design guidelines to flow will be focused on AgentSheets, since it is the system we have the most experience with. The following section will provide a brief introduction to AgentSheets sufficient to understand the design guidelines.

2. AGENTSHEETS

AgentSheets (Ioannidou & Repenning, 1999; Alexander Repenning & Ioannidou, 1997; Alexander Repenning, Ioannidou, & Ambach, 1998; Alexander Repenning & Sumner, 1995) initially grew out of the idea of building a new kind of computational media that allows casual computer users to build highly parallel and interactive simulations, replacing simple numbers and strings of spreadsheets with autonomous agents. The simulations are used to communicate complex ideas or to simply serve as games. Adding a spreadsheet paradigm to agents enabled the manipulation of large numbers of agents and, at the same time, organize them spatially through a grid. An early prototype of AgentSheets was built in 1988 to run on the Connection Machine (a highly parallel computer with 64000 CPUs).

Partially influenced by the spreadsheet paradigm, the AgentSheets agents were designed to feature a rich repertoire of multimodal communication capabilities. Users should be able to see agents and to interact with them. As more communication channels became available in mainstream computers, they got added to the repertoire of agent perception and action. Text-to-speech allowed agents to talk and speech recognition allowed users to talk to their agents. When the Web

started to gain momentum agents got extended to be able to read and interpret Web pages (Figure 2).

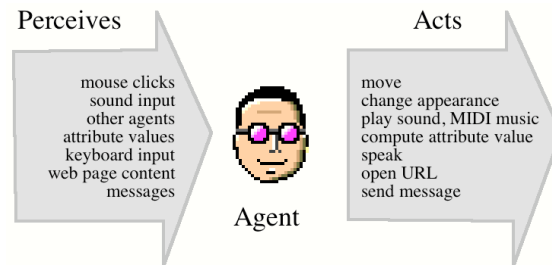


Figure 2. *AgentSheets' includes multimodal Agents that can perceive and act*

An agentsheet (also called a worksheet) is a grid-structured container of agents. In contrast to a spreadsheet each cell in an agentsheet may contain any number of interacting agents stacked on top of each other. The grid allows agents to employ implicit spatial relations, e.g., adjacency, to communicate with other agents. Table 1 shows examples of agentsheets used for a variety of applications. End-user development consists of creating these applications by defining agent classes including the definitions of the agent looks, i.e., iconic representations, as well as the behaviour of agents.

The programming part of End-User Development lies in the specification of agent behavior. Agent behaviors are rule-based using a language called Visual AgenTalk (VAT) (Alexander Repenning & Ambach, 1996a, 1996b; Alexander Repenning & Ioannidou, 1997). Visual AgenTalk is an end-user programming language that has emerged from several iterations of design, implementation and evaluation of previous AgentSheets programming paradigms including programming by examples using graphical rewrite rules, and programming by analogous examples.

Visual AgenTalk rules are organized as methods including a trigger defining when and how a method will be executed. Figure 3 shows a traffic light agent cycling between green, yellow and red. The first method called "While Running" will be triggered one every simulation cycle. A rule can have any number of conditions and actions. The only rule of the "While Running" method checks time and calls another method called "Switch" every 3 seconds. The "Switch" method will advance the traffic light to the next state. The next color is selected based on the current state of the traffic light. Details on how end-users program in VAT are discussed in the design guidelines.

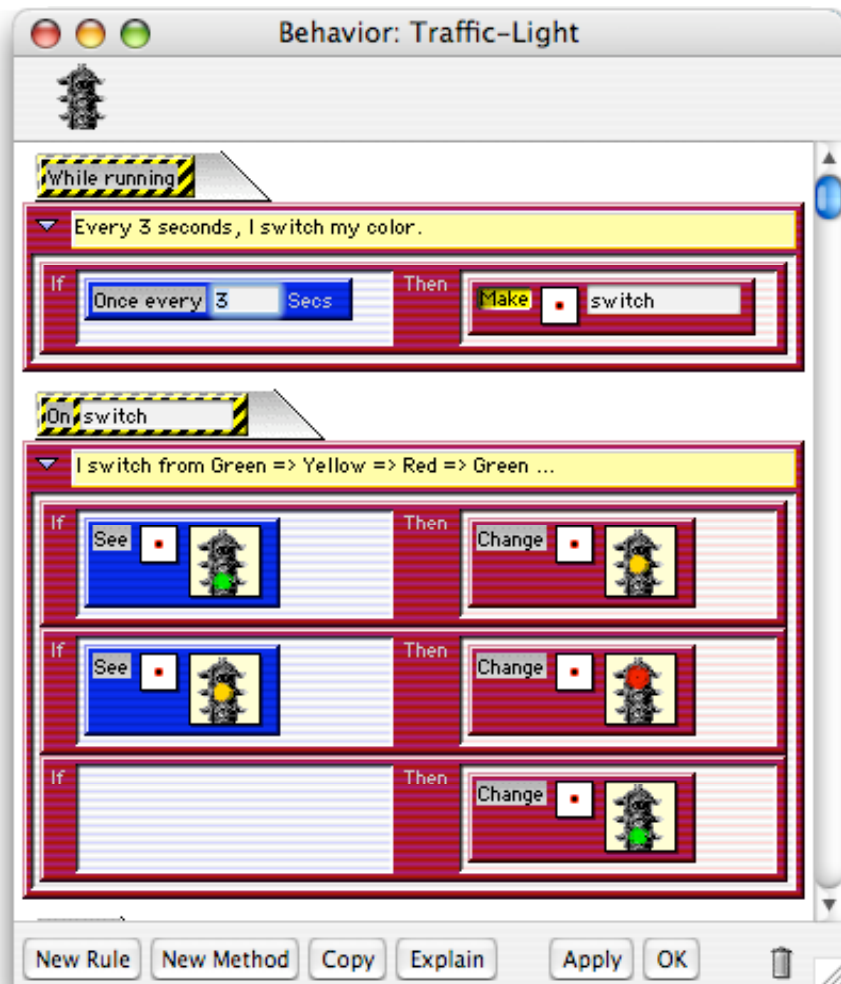


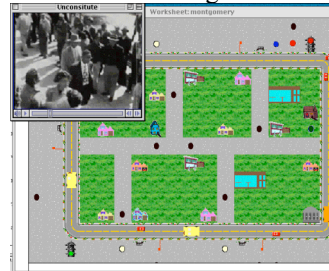
Figure 3. A Visual AgenTalk Behavior Editor: Rules can have any number of conditions and actions. Rules are grouped into methods including triggers defining when and how methods will be invoked.

K-12 Education: Elementary School



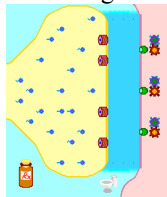
Collaborative Learning: Students learn about life science topics such as food webs and ecosystems by designing their own animals. The AgentSheets Behavior Exchange is used to facilitate collaborate animal design. Groups of students put their animals into shared worlds to study the fragility of their ecosystems.

K-12 Education: High School



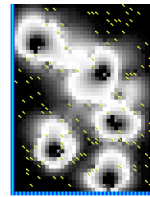
Interactive Story Telling: History students create interactive stories of historical events such as the Montgomery bus boycott.

Training



Distance Learning: With SimProzac patients can explore the relationships among Prozac, the neurotransmitter serotonin, and neurons. By playing with this simulation in their browsers, patients get a better sense of what Prozac does than by reading the cryptic description included with the drug.

Scientific Modelling



Learning by visualization and modeling: The effects of microgravity onto E.coli bacteria are modelled by NASA. This is a simulation of an experiment that was aboard the Space Shuttle with John Glenn. This simulator requires several thousand agents.

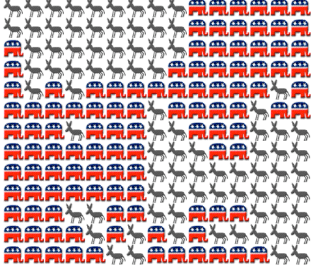
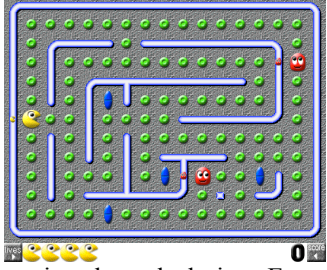
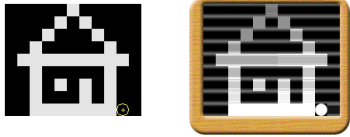
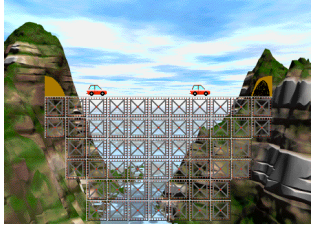
<p style="text-align: center;">Educational Games</p>  <p>Learning through simulation use: This simple voting simulation explains concepts such as clustering, migration and stability of two party systems. Can it predict the outcome of the election in 2000?</p>	<p style="text-align: center;">Non-Educational Games</p>  <p>Learning through design: Even if the finished simulation/game is not directly related to educational goals, the process of building the simulation may be very educational. The Ultimate Pacman is a complete game based on complex Artificial Intelligence algorithms and the non-trivial math of diffusion processes.</p>
<p style="text-align: center;">Interactive Illustrations</p>  <p>How does a TV work? This simulation illustrates how a picture is scanned in by a camera (left), transmitted to a TV set and converted back in to a picture (right). Users can paint their own pictures and play with TV signal processing parameters.</p>	<p style="text-align: center;">Deconstruction Kits</p>  <p>Learning by taking apart: What makes a bridge stable? The goal presented to the users of this simulation is to remove as many elements of the bridge as possible without making the bridge collapse. A number of connected issues are revealed including forces, architecture, and geometric perspective. This simulation was featured on the PBS Mathline.</p>

Table 1. *AgentSheets Examples*

This minimalist introduction to AgentSheets is only provided to give the reader on a quick sense on what AgentSheets is and what type of applications it has been used for. The focus of this chapter is to provide design guidelines that may help designers building end-user development systems. These design guidelines are intended to provide prescriptive descriptions of End-User Development suggestions. These guidelines have emerged from observing people using the AgentSheets system. The guidelines are generalized as much as possible to help designers of systems that have nothing to do with simulation authoring and programming environments for kids. Nonetheless, AgentSheets examples are included to provide sufficient substance illustrating concrete problems. Our hope is that while concrete manifestations of problems may change over time (e.g., new versions of operating systems, GUIs) the guidelines will still hold. In contrast to design patterns (Alexander et al., 1977; Gamma, Helm, Johnson, & Vlissides, 1995), the design guidelines not only observe existing patterns but provide descriptive instructions in form of implementation examples. For simpler reference, we have categorized the design guidelines into syntactic, semantic and pragmatic. Finally, all guidelines are presented with optimal flow in mind. Many guidelines are reactions to breakdowns where users reacted with either anxiety or boredom.

This list of design guidelines is not exhaustive. No set of design guidelines can guarantee the design of a successful end-user development system. The notion of flow can help us – to a certain degree – to design systems that can be learned. The following three sections will present guidelines at the syntactic, semantic and the pragmatic level.

3. SYNTACTIC GUIDELINES

Syntactic problems of traditional languages, e.g., the frequently mentioned missing semicolon in programming language such as Pascal or C, pose a challenge to most beginning programmers for no good reason. This quickly leads to anxiety without contributing much towards the conceptual understanding of a programming language. A number of end-user but also professional programming environments have started to address this problem.

Visual programming (Burnett, 1999; Burnett & Baker, 1994.; Glinert, 1987; Smith, 1975) is one such paradigm that attempts to pictorially represent language components that can be manipulated to create new programs or to modify existing ones. Visual programming languages are “a concerted attempt to facilitate the mental processes involved in programming by exploiting advanced user interface technology” (Blackwell, 1996). For instance, the visual representation of the language components and constructs often eliminates the need to remember syntax. Professional programming, not geared towards end-users, is following by using approaches that range from making syntactic errors hard to impossible.

3.1 *Make syntactic errors hard*

Syntax coloring of reserved words and symbols in traditional programming language environments helps the programmer's perception, which in turn helps in creating syntactically correct programs. Symbol completion in languages such as Lisp, and more recently C, helps programmers to produce complete and correct symbols already defined in the programming language, minimizing the possibility of making a typographical error and therefore a syntax error. Finally, wizards utilizing templates for defining program attributes and then generate syntactically correct code, such as the wizard in the CodeWarrior programming environment, syntactically support programmers.

3.1.1 *example: Apple Xcode development environment*

Apple's Xcode developer tool set includes highly customizable syntax coloring (Figure 4) combined with code completion that can be manually or automatically invoked by a user. Project templates will generate boilerplate code for the user when creating certain types of projects.

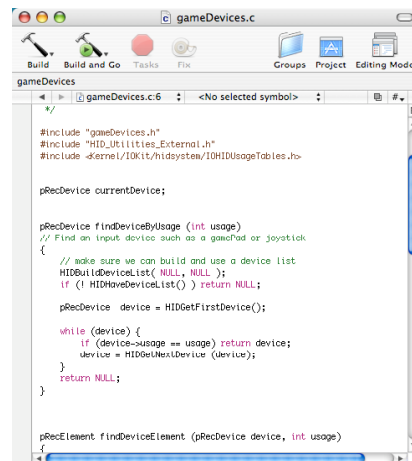


Figure 4. *Syntax Coloring in Apple's Xcode Programming Tools*

Even with this type of support these tools are only marginally useful to end-users since they still pose a high challenge and require sophisticated skills such as the ability to create programs in C.

3.2 *Make syntactic errors impossible*

Other programming approaches employ mechanisms that help with the syntactic aspects of programming in such a way that syntactic errors are essentially impossible.

3.2.1 example: Programming by Example

In Programming by Example (PBE) (Cypher, 1993; Lieberman, 2001) syntactic problems are avoided by having the computer, not the user, generate programs. A computer observes what the user is doing and, for repetitive tasks, learns them and ultimately does them automatically.

An instantiation of the PBE approach is found in AgentSheets' Graphical Rewrite Rules (GRR) (Bell & Lewis, 1993). In GRR the program is the result of manipulating the world. As the user interacts with the computer world, the PBE system observes users and writes the program for them. The only skill users need to have is the skill to modify a scene. To program a train, the user creates examples of how trains interact with their environment (Figure 5). The fact that trains follow train tracks is demonstrated by putting a train onto a train track, telling the computer to record a rule, and moving the train along the train track to a new position. The computer records this user action as a pair of Before/After scenes. The generalize the rule users will narrow the scope of the rule to the required minimum and, if necessary, remove irrelevant objects. A tree next to the train track is likely to be irrelevant and consequently should be removed from the rule, whereas, a traffic light could be essential to avoid accidents.

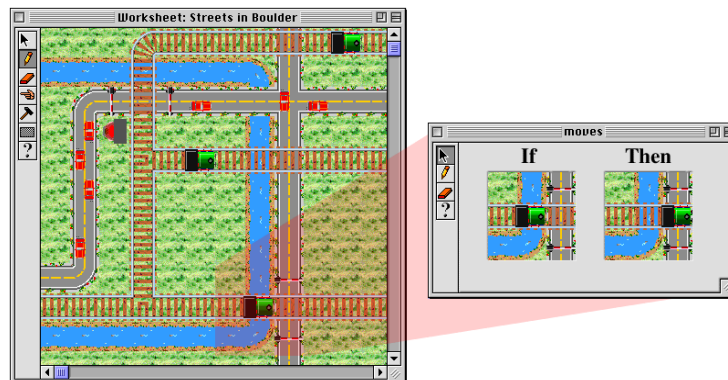


Figure 5. Programming by Example through graphical rewrite rules.

Programming by Example works well for end-user development from a flow perspective, especially at the beginning. Skills and challenges are well balanced because programming is achieved by merely manipulate scenes (moving, deleting, and creating objects). However, many programming by example systems do not scale well with problem complexity. Two scenarios are possible.

- 1) Getting bored: Assume the user wants to create Conway's "Game of Life". A first concrete rule is built quickly but then, in order to implement the entire condition of n out of m cells alive the user is forced to create all $f(n, m)$ rules since the system cannot generalize. Using only a medium amount of skill for basically no challenge (as all rules are simple variants of the first existing one), the user becomes bored.

- 2) Getting anxious: Assume the user wants to create a situation where numerical diffusion is necessary – for example to illustrate how heat diffuses in a room. The pure iconic nature of Graphical Rewrite Rules makes them ill suited for implementing such numerical problems. The mismatch of the language and the problem makes the task of implementing diffusion complex. If the complexity of the task increases in a way that the main programming paradigm gets exhausted, the user is expected to learn something new that cannot easily be connected or does not fit to the current paradigm. As a consequence, the challenges soar up disproportionately. The learning curve is no longer a gentle slope and the end-user programmer leaves the optimal flow area of the graph and ends up in a state of anxiety (Figure 1).

In either of these two cases, the programming paradigm worked well for a short learning distance, keeping challenges in proportion to skills. But the paradigm reaches a critical threshold at which either the challenges go way up or the existing skills are no longer well used. We called this effect “trapped by affordances” (Schneider & Repenning, 1995).

3.3 *Use Objects as Language Elements*

Instead of just representing programming language elements as character strings – the way it is done in most professional programming language such as C or Java – they can be represented as complete objects with user interfaces. Visual representations of these objects (shapes, colors, and animation) may be selected in ways to strongly suggest how they should be combined into a complete working program. Drag and drop composition mechanisms including feedback functions can be employed to guide users. Additionally, language elements may embody user interfaces helping users to define and comprehend parameters of language elements.

3.3.1 *example: Puzzle Shape Interfaces*

One approach to achieve easy program composition was languages that use a compositional interfaces for assembling programs from language components with visual representations. Glinert’s BLOX Pascal uses flat jigsaw-like pieces that can be snapped together (Glinert, 1987). The BLOX method was one of the first proposals on using the third dimension for visual programming. While this approach alleviates some syntactic issues such as correct sequencing of language statements and parameter setting, the BLOX Pascal language is still, in essence, a professional programming language. End-user programming languages have been developed with the same philosophy. LEGO Mindstorms includes an end-user programming environment for kids to program the LEGO RCX Brick, which can be used to control robotic LEGO creations. The language used to program the Brick is LEGO RCX Code, which uses a jigsaw puzzle user interface similar to BLOX Pascal.

3.3.2 example: AgentSheets Visual AgenTalk

The AgentSheets simulation environment the Visual AgenTalk language (Alexander Repenning & Ambach, 1996a, 1996b; Alexander Repenning & Ioannidou, 1997). All language elements (conditions, actions and triggers) in VAT are predefined and reside in palettes (Figure 6). Using drag and drop, users essentially copy these language elements and assemble them into complete programs, in if-then forms in an agent's behavior editor, such as the one shown in Figure 3.

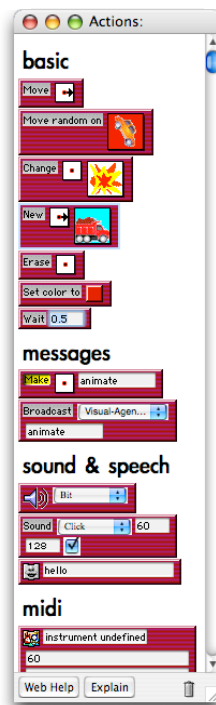


Figure 6. *The Visual AgenTalk action palette*

Command parameters, called “type interactors” in AgentSheets, are set by the user via direct manipulation. For instance, the SEE condition tests for the presence of an agent with a certain depiction in a specific direction (Figure 7). Direction and depiction are parameters to the command and are both set via direct manipulation. This is important because, as pointed out by Nardi (Nardi, 1993), the way parameters are specified can affect the extent to which the programmer must learn language syntax. The integration of parameters that are directly manipulatable, such as the 2D pop-up dialogs for direction and depiction, elevate the program onto the level of a user interface combining ideas of form-based interfaces (Nardi, 1993) with end-user programming.

In terms of flow, an initial price needs to be paid because the end-user is forced

to explicitly deal with programming language constructs. Direct manipulation interfaces help to avoid syntactic problems. A form-based interface includes iconic representations of object created by the end-user (e.g., drawings of agents). Depending on the repertoire of the end-user programming language this approach is likely to be more expressive compared to programming by example approaches by allowing the end-user to combine language constructs in way that could not have been anticipated by a PBE approach.

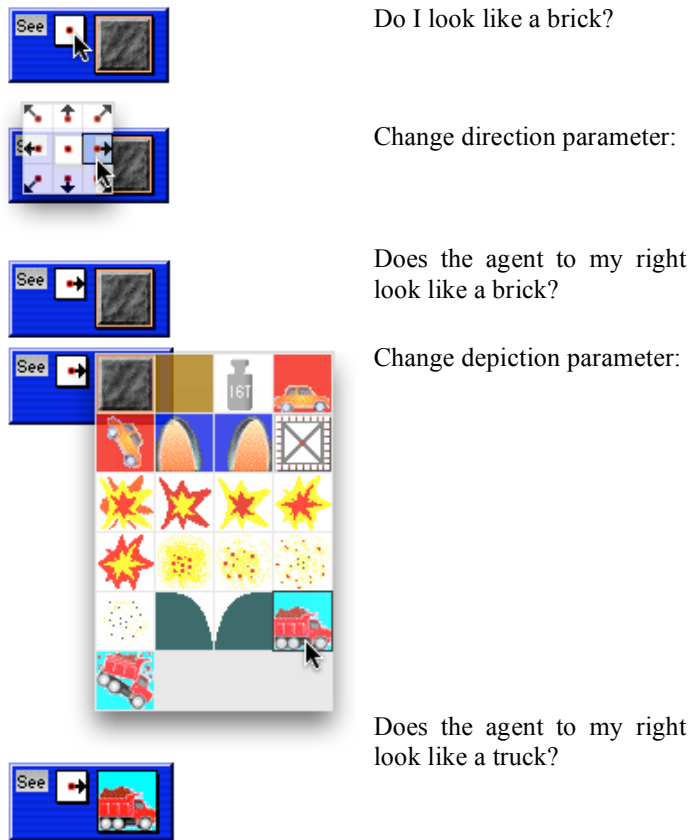


Figure 7. *SEE* condition and its parameters

4. SEMANTIC GUIDELINES

The reduction of syntactic problems is a necessary but not sufficient goal of end-user development. The frustration over missing or misplaced semicolons in professional programming has probably put an early end to many programming

careers. However, a program that is syntactically correct not necessarily an efficient, working, or meaningful program. Support at the semantic level helps end-users with bridging the conceptual gap between problem and solution.

4.1 *Make domain-oriented languages for specific EUD*

The elements of a programming language are often close to the elements of a natural language. Using some form of syntax, programming language elements representing words such as nouns, verbs, and adjectives are aggregated into complete statements. In early, low level, programming languages these words were typically heavily inspired by the domain of computers. In assembly programming nouns refers to elements found on the CPU such as registers and verbs refer to instructions at the CPU level such as “load”. Modern high-level languages have replaced references to low-level technological details with more general concepts such as data structures (e.g., integer, floats, structures, objects). But even this level is difficult for end-user developers. Correlating concepts relevant to a certain application with elements of a generic language can be a large challenge. Nardi suggest the use of task-specific programming languages (Nardi, 1993) as means to reduce this correlation effort. Along a similar vein Eisenberg and Fischer postulate the use of domain-oriented programming languages (Eisenberg & Fischer, 1994). A good example of such a domain-oriented programming language was the pinball construction kit allowing people not only to play a simulated pinball game but also to build their own. The pinball construction kit language consisted of a palette of pinball components such as flippers and bouncers.

With an extensible architecture, AgentSheets was used to build a number of domain-oriented languages such as the Voice Dialog Design Environment (Alex Repenning & Sumner, 1992), the AgentSheets Genetic Evolutionary Simulations (Craig, 1997) and the EcoWorlds environment by the Science Theater team (Brand, Rader, Carlone, & Lewis, 1998; Cherry, Ioannidou, Rader, Brand, & Repenning, 1999; Ioannidou, Rader, Repenning, Lewis, & Cherry, 2003). Domain-orientation can dramatically reduce the challenge of using a tool but at the same time reduces the generality of a tool. In terms of flow, this translates into highly specific environments that require little training, but have a limited range of applications.

4.1.1 *example: EcoWorlds*

The EcoWorlds system, a domain-oriented version of AgentSheets for ecosystems, was used as a life sciences learning tool in elementary schools. A learning activity would consist of students working in teams to create artificial creatures participating in an ecosystem. Large predators can eat small animals that may eat even smaller animals or plants. The goal of this activity was for students to understand the fragile nature of ecosystems. They had to tweak their designs very carefully to create stable environments. A first attempt of the project tried to use AgentSheets and KidSim (Smith, Cypher, & Spohrer, 1994) (later called Creator). The challenge of mapping domain concepts relevant to the curriculum such as reproduction rates and food dependencies was simply too much of a challenge for kids to achieve. The Science

Theater team created a domain-oriented version of AgentSheets (called EcoWorlds) to capture the domain of ecosystems. A trivial example of this process was the replacement of the Erase action with the Eat action. More complex language elements included complete templates referring to reproduction rates, food categories and other ecosystem-specific concepts.

With EcoWorlds, kids were able to create complex, and most importantly, working ecosystem simulations. There are trade offs, of course. The design of a well-working domain-oriented language is by no means trivial. Many domains tend to change over time requiring maintenance of the domain-oriented programming language. The conceptualization of a domain by the language designers may not match the conceptualizations of the same domain by the users. User-centered design (Lewis & Rieman, 1993; Norman, 1986) can help to some degree. Finally, the specificity of a language to one domain may render it useless to other, non-related domains.

4.2 Introduce Meta-Domain Orientation to Deal with General EUD

A more general-purpose EUD language can make few, if any, assumptions about the application domain. Consequently, the usefulness of a domain-oriented language is severely limited. Meta-Domain oriented languages are languages that are somewhat in between the application domain and the computer domain. Spreadsheets are examples of meta-domain orientation. The spreadsheet metaphor, while inspired originally by bookkeeping forms, is a neutral form of representation that is neither directly representing the application domain nor is a low-level computer domain representation. People have used spreadsheets for all kinds of applications never anticipated by the designers of spreadsheet tools. More specific applications that initially were solved with spreadsheets, such as tax forms, have meanwhile been replaced with complete, domain-oriented tools such as tax-form tools. Where domain-orientation is possible and effective (from the economic point of view, e.g., if there are enough people with the exact same problem) domain-oriented tools are likely to supersede more generic tools in the long run.

AgentSheets uses its grid-based spatial structure as meta-domain allowing people to map general problems onto a spatial representation. A grid is an extremely general spatial structure that can be employed to represent all kinds of relationships. In some cases, the grid is employed to spatially represent objects that also have a natural spatial representation. In other cases, the grid is used to capture conceptual relationships that have no equivalence in the physical world.

Meta-domain orientation manifests itself in the End-User Development language. In AgentSheets the Visual AgenTalk language includes a variety of spatial references embedded at the level of commands and type interactors.



Actions: Some VAT actions are intrinsically spatial. For instance, the Move action is used to make an agent move in space from one grid location to another adjacent position. Others actions are spatial in conjunction with their parameters. The parameter of the Erase action defines where to erase an agent.

Conditions: A large number of VAT conditions are used to evaluate spatial relationships. The See condition allows an agent to check if an adjacent cell in a certain direction contains an agent with a certain depiction.

Type Interactors: Type Interactors are parameters of VAT condition/action commands including a user interface. Some type interactors such as Direction-Type allow users to select a direction to an adjacent grid location.

Table 2: *Visual AgenTalk Actions, Conditions and Type Interactors*

4.3 Use Semantic Annotations to Simplify the Definition of Behavior

End-User Development is not limited to programming. It includes the creation and management of resources such as icons, images, and models. The program defining the behavior of an application needs to be connected with resources defining the look of an application. End-User Development tools should support the creation as well as the maintenance of these connections. At a simple, syntactic level, this connection should become visible to a user. Visual AgenTalk, for instance, includes the Depiction-Type interactor, which essentially is a palette of all the user-defined icons. Things get more complex at the semantic level because development systems cannot automatically derive semantic information from artwork. However, with a little bit of semantic annotation or meta-data provided by users an end-user development system can greatly simplify the development process.

4.3.1 Example: Semantic Rewrite Rules

Graphical rewrite rules, as a form of end-user programming, suffer from their implicit underlying model. Interpretation of rewrite rules limited to syntactic properties makes it laborious for end users to define non-trivial behavior. Semantically-enriched graphical rewrite rules have increased expressiveness, resulting in a significantly reduced number of rewrite rules. This reduction is essential in order to keep rewrite rule-based programming approaches feasible for

end-user programming. The extension of the rewrite rule model with semantics not only benefits the definition of behavior but additionally it supports the entire visual programming process. Specifically the benefits include support for defining object look, laying out scenes consisting of dependent objects, defining behavior with a reduced number of rewrite rules, and reusing existing behaviors via rewrite rule analogies. These benefits are described in the context of the AgentSheets programming substrate.



Figure 8. *Connectivity Editor*. Users add semantic annotations to define the meaning of an icon. A horizontal road connects the right side with left side and the right side with the left side.

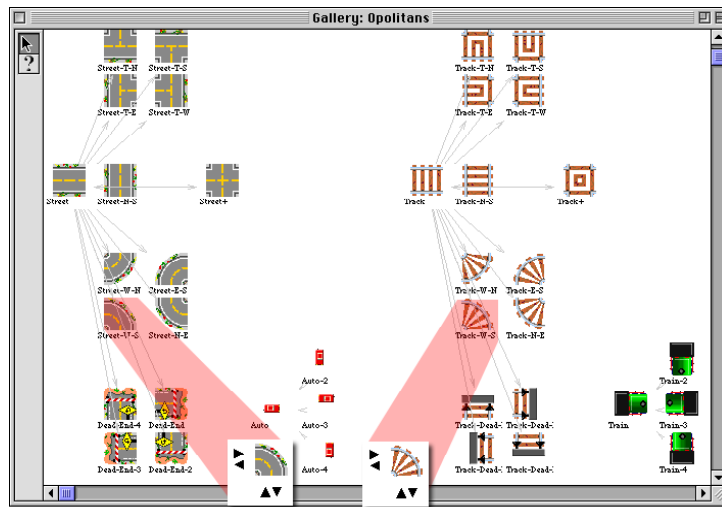


Figure 9. *Icons and their semantic annotations are geometrically transformed. Semantic information is used to establish isomorphic structures for generalized rules and analogies.*

Semantic Rewrite Rules (Alexander Repenning, 1995) allow users to annotate their icons with semantic information such as connectivity. For instance, an icon representing a horizontal strip of road can be annotated with connectivity arrows indicating that this road connects the right with the left and the left with the right. AgentSheets can then transform these icons syntactically as well as semantically.

The syntactic transformation will bend, rotate, split, and intersect icons by applying bitmap operations to the original road icon (Figure 8). The semantic information will be transformed by automatically deriving the connectivity information of the transformed icons. Finally, the single rewrite rule describing how a train follows a train track (Figure 5) is now interpreted on a semantic level. This one rule is powerful enough that the train can follow any variant of train tracks without the need to create the large set of all the permutations of trains driving in different directions and train tracks. In terms of flow, the user can now, with the same degree of skill, tackle substantially larger challenges.

4.3.2 Example: Programming by Analogous Examples

Analogies are powerful cognitive mechanisms for constructing new knowledge from knowledge already acquired and understood. When analogies are combined with programming by example, the result is a new end-user programming paradigm, called Programming by Analogous Examples (Alexander Repenning & Perrone, 2000; Alexander Repenning & Perrone-Smith, 2001), combining the elegance of PBE to create programs with the power of analogies to reuse programs.

This combination of programming approaches substantially increases the reusability of programs created by example. This merger preserves the advantages of programming by example and at the same time enables reuse without the need to formulate difficult generalizations. For instance, if programming by example is used to define the behavior of a car following roads in a traffic simulation, then by analogy this behavior can be reused for such related objects as trains and tracks by expressing that “trains follow tracks like cars follow roads”.

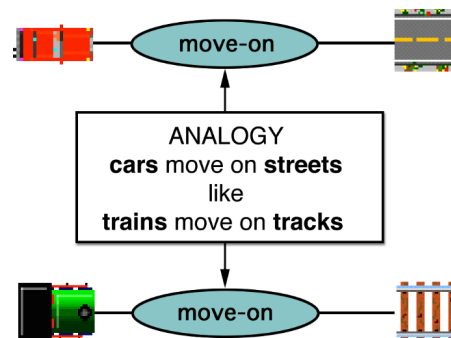


Figure 10. An analogous example defines the interactions between cars and roads by establishing an analogous connection to trains and train tracks.

The analogy between cars and trains can be established because of the semantic information. At the syntactic level the system could not have recognized the relationships between cars and trains but the semantic information is sufficient to allow the system to find the corresponding match based on the connectivity of icons. In terms of flow, Programming by Analogous Examples simultaneously reduces the

challenge of programming and requires few skills to establish an analogy. However, sometimes analogies can be hard to see and even when they can be applied analogies may break down requiring some kind of analogy exception handling.

5. PRAGMATIC GUIDELINES

In addition to the syntactic and semantic support described above, a programming language has to provide pragmatic support to be effective as an end-user development tool kit. That is, an end-user development language should make programs personally relevant to the end-user and the programming process more practical.

5.1 *Support Incremental Development*

When a programming language allows and supports incremental development of programs, end-user programmers do not feel that what they are asked to do with the computer is too difficult, but instead that they are building up the necessary skills in an incremental fashion, thus staying in the optimal flow of the learning experience. Incremental development provides instant gratification, avoids leaps in challenge and allows skills to grow gradually.

- Get instant gratification: end-user programmers have the opportunity to execute and test partially complete programs or even individual language components, getting feedback and gratification early on in the programming process.
- Avoid leaps in challenge: the step-by-step creation of a program enables end-user programmers to incrementally add complexity to their program, avoiding huge leaps in challenge and tasks that would otherwise be infeasible and would undoubtedly lead to anxiety (Figure 1). Traditional languages that force the programmer to have a complete program-compile-run cycle before each test of the program are typically more time-consuming and drive programmers into a style of programming where they write big chunks of code before trying it out, which often makes debugging harder.
- Allow skills to grow gradually: when end-users incrementally develop, test, and debug programs, their skills grow in proportion to the challenge they face. Incremental development provides the end-user programmers with mechanisms to first tackle small parts of the challenge before incorporating them to the bigger picture.

The form of exploratory and experimental programming that is afforded by small increments of code is well suited to end-user programmers that are not experienced programmers and have not received formal education in software design methods and processes.

5.1.1 *example: Tactile Programming*

AgentSheets' Visual AgenTalk is an end-user programming language that elevates the program representation from a textual or visual representation to the status of a

user interface. In its elevated form, the program is the user interface. By providing language objects (conditions and actions) packaged up with user interfaces, VAT is rendered into a tactile programming language.

Tactility is used here not in the sense of complex force feedback devices that are hooked up to computers, but much more in the sense used by Papert to explain the closeness of bricoleur programmers to their computational objects (Papert, 1993). One departure from Papert's framework is that the notion of computational objects in Visual AgenTalk is not limited to the objects that are programmed, such as the Logo turtle, but also applies to the programming components themselves, which are elevated to the level of highly manipulatable objects (Alexander Repenning & Ioannidou, 1997).

Visual Programming is employing visual perception to simplify programming by increasing the readability of programs. Tactile Programming does not question this goal, but hopes to make programming more accessible to end-users by adding the perception of manipulation to visual perception. In Tactile Programming, programs are no longer static representations nor is the notion of manipulation reserved to only editing programs. Instead, tactile programs and their representations are dynamic and include manipulation, such as setting parameters by manipulating parameter spaces (Figure 11) or composing programs by dragging and dropping languages pieces in behavior editors.

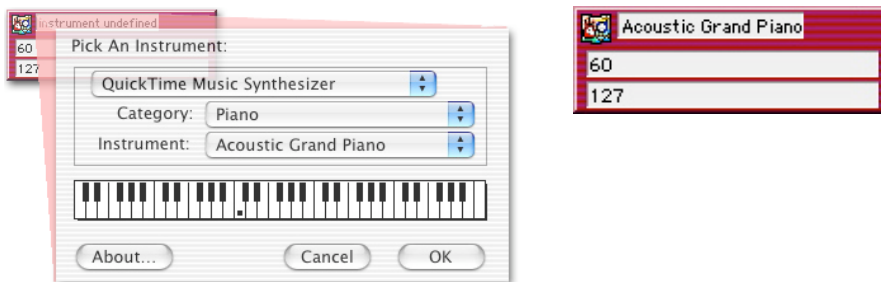


Figure 11. *The function of a tactile programming object can be perceived through interaction.*

Tactile Programming primitives and programs not only have enhanced visual representations to help program readability, but also have interactive interfaces to assist with program writability. With Tactile Programming programs can be composed incrementally along clearly defined boundaries, making program composition easy.

5.2 Facilitate Decomposable test units

Traditional programming languages do not allow out-of-context testing of individual statements, because for instance there may be undefined variables in that small

fragment of code to be tested. In contrast, the kind of interface tactile programming provides, supports an exploratory style of programming, where users are allowed to “play” with the language and explore its functionality. Perception by manipulation afforded by tactile programming allows end-users to efficiently examine functionality in a direct exploration fashion. Any VAT language component at any time can be dragged and dropped onto any agent. The component will execute with visual feedback revealing conditions that are true or false and showing the consequences of executed actions.

Program fragments can be tested at all levels: conditions/actions, rules, and methods. For instance, dragging the move command from the action palette onto the Ball agent in the worksheet will make the ball move to the right (Figure 12).



Figure 12. *What does move right do? Drag program fragment onto agent to see consequences. Dragging move-right action onto ball will make ball move to the right one position.*

Condition commands, when dragged and dropped onto agents, will reveal whether the condition holds for the agent in its current context. If the See condition is dragged onto the soccer player in the worksheet, visual and acoustic feedback will immediately indicate that this condition would not hold. In the case of Figure 13, it will indicate that the condition is “true”.



Figure 13. *Testing conditions: dragging See-right-ball condition onto soccer player agent will test if there currently is a ball immediately to the right. The condition is true.*

Dragging and dropping an entire rule onto an agent will test the entire rule. Step-by-step with visual feedback, all the conditions are checked. In our example rule (Figure 14), only one condition exists. If the soccer player agent sees to his right a

ball agent, the condition is successfully matched and, as consequence, all the actions are executed— in this case, changing the depiction of the player (Change action), colorizing him red to show who is in control of the ball (Set color to action) and keep that for a while (wait action); then tell the ball (which happens to be to his right) to run its “kick-right” method and reset the colorization back to its original colors. The results of the executed rule are graphically shown on the right (Figure 14). Had the condition of that rule failed, acoustic feedback would have been provided and the condition that failed would have blinked to indicate the problem.

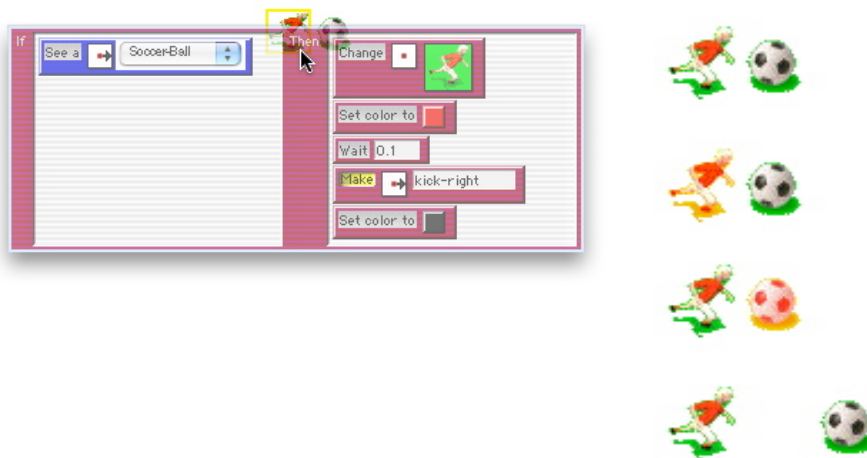
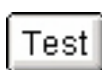


Figure 14. *Testing Rules. If all the conditions of the rule are true then the actions will be executed. One action after another get highlighted, and the consequence of running it visualized in the simulation. The agent changes to look like the kicking player, it turns red, after some time it sends the kick-right message to the ball, and turns its color back to normal.*

Tactile programming with decomposable test units at different levels of granularity of the programming language (individual commands, rules, methods) provides easy debugging even for end-user programmers that do not possess the skills of professional programmers in debugging.

On the down side, drag and drop may not necessarily be the best mechanism for testing these language components. While drag and drop is an excellent mechanism for program composition, for this type of testing it may not be as intuitive or obvious as one may think. User testing has shown that when using the Macintosh version of the AgentSheets simulation-authoring tool, users have to be told this drag and drop testing feature exists. This was remedied in the Windows version of the software by



adding a Test button in the behavior editor window. Instead of dragging and dropping commands or rules onto agents in the worksheet, a user simply selects the language component to be tested and the agent on

which to test it on and presses the Test button. Not only this makes this debugging feature more apparent, but it also affords multiple tests of the same language component without the mundane effort of dragging and dropping the same piece over and over again.

Whatever the mechanism, the point remains that end-user programming languages need to allow their users to test any piece of the program at multiple granularities (command, rule, method) in the context of an agent in the worksheet at any time during the development process. This supports understandability of the language and therefore enhances program writability.

5.3 *Provide Multiple Views with Incremental disclosure*

One of the criticisms of visual programming languages is that they use a lot of screen real estate (Green & Petre, 1996). To improve program readability and consequently program comprehension, multiple views of the same program should be available to end-user programmers. Using disclosure triangles is one technique to collapse and expand program components.

5.3.1 *example: Disclosure Buttons and Disclosure Triangles*

In AgentSheets, disclosure triangles are used to show or hide methods in an agent's behavior editor. Figure 15 shows the collapsed "Advance" method of the Car agent with only its name, documentation, and number of rules contained visible in the editor.

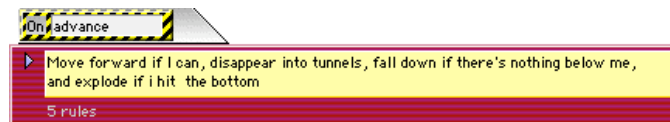


Figure 15. *A collapsed method shows only the documentation information and, as indicator for method complexity, the number of rules contained.*

Figure 16 shows the expanded version of the same method with all the rules exposed.

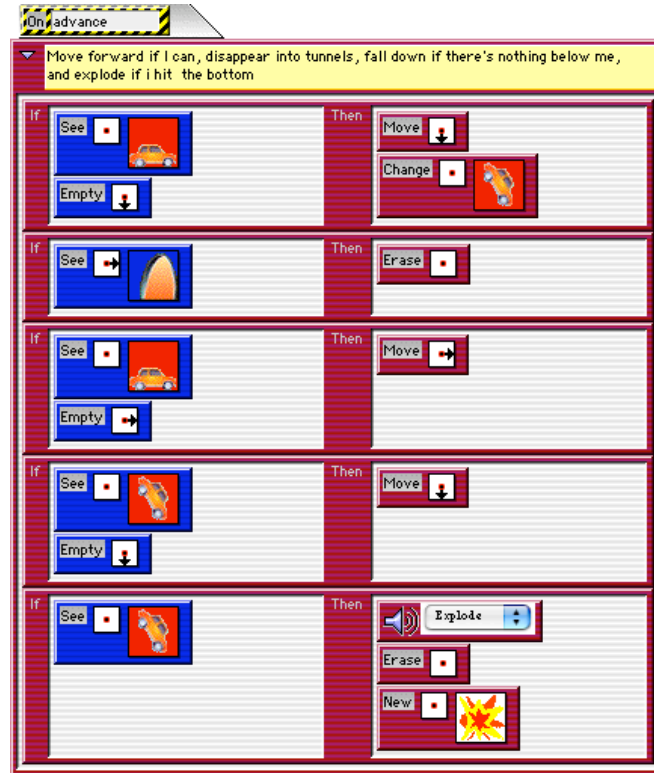


Figure 16. *The expanded view provides access to all the rules.*

In terms of flow, the ability to switch between views helps to manage information clutter and consequently simplifies the location of relevant information.

5.4 Integrate Development Tool with Web Services

The Web is a rich resource of information that can help the design process. Development tools in general and end-user development tools specifically should make use of these resources by providing seamless connection mechanisms helping to find relevant resources based on the current design state.

5.4.1 Example: Design-Based Google Image Search

AgentSheets can locate relevant artwork based on the design state of an agent using the Google image search. Say a user creates an agent called “red car” and designs an icon quickly using a bitmap editor. Instead of creating their own artwork users may use AgentSheet’s “Search Depictions on Web” function. This function



will use information from the design environment, the name of the agent class “red car”, compute a Google query and open a Web browser. There is no guarantee that a suitable image is found but if users do find a good match they can import images into AgentSheets.

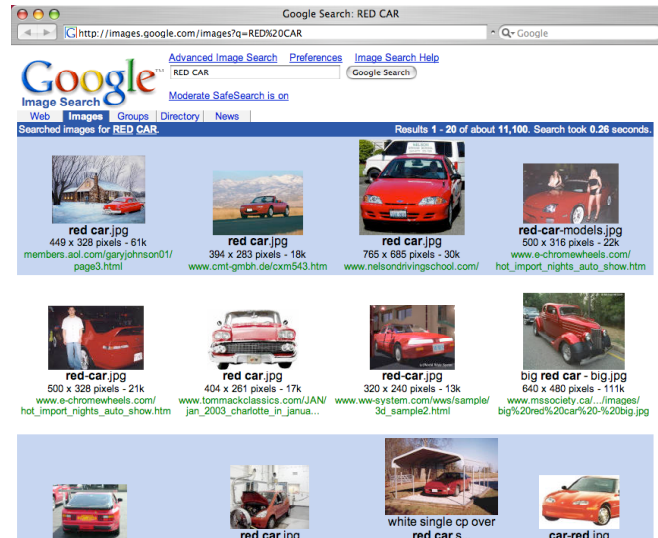


Figure 17. A Goggle Image search triggered by the Design of a "red car" agent. Suitable images can be imported into AgentSheets.

Integration with Web services are relevant to flow, in the sense that integration not only extends design spaces with external resources but also reduces rough transitions between otherwise non-connected tools.

5.5 Encourage Syntonicity

Papert used the term syntonicity to describe how children's physical identification with the Logo turtles helped them more easily learn turtle geometry (Papert, 1980, 1993; Papert & Harel, 1993). Syntonicity helps people by allowing them to put themselves into the “shoes” of the objects they create and program. As a mindset syntonicity encourages the development of mini scenarios helping to disentangle potentially complex interaction between multiple objects: “If I where the car driving on the bridge doing ...” As an extension to Papert’s stance towards syntonicity, we found that syntonicity can be actively cultivated by a system through a number of mechanisms. Moreover, we find syntonicity relevant to end-user development in general because it helps to overcome some essential misconceptions of programming often found in non-programmers.

5.5.1 *example: First-Person Domain-oriented language components*

Students often find it difficult to map their ideas about science phenomena onto the operations provided by a visual language (Brand & Rader, 1996). Commands that match the problem domain can simplify this process and also focus the students' attention on important aspects of the content. The Visual AgenTalk language has been customized by researchers at the University of Colorado conducting research on modeling ecosystems in elementary school settings to support the programming of concepts in that domain. A number of domain-oriented commands were introduced to support the definition of predator-prey interactions. For example, rules that enable a predator to eat are stated as, "If I can select food <description of prey, based on features> then try to eat it, because I am <description of self, specifying why I can eat this prey>." This set of commands replaces more basic actions, such as "see" and "erase," with the specific actions of selecting food and trying to eat it. The design of the commands also requires students to enter features of the predator and prey, thereby reinforcing science ideas about structure and function (Brand et al., 1998; Cherry et al., 1999; Rader, Brand, & Lewis, 1997; Rader et al., 1998).

Not only were these customized commands domain-oriented, but they were also presented in the first person, e.g. "I eat". The result was for students to identify with the agents they were building (namely, the animals), which was apparent in their lively discussions about their ecosystem simulation. The students typically referred to their animals in the first person, saying for example "I'm dead" rather than "My animal is dead," or "I can eat you" rather than "my Ozzie can eat your Purple Whippy Frog." Perhaps because of this identification, students were very motivated to ensure that their animals survived. Although students initially had a tendency to want their animals to survive at the expense of other populations, this tendency was often mitigated once they realized that the other species were necessary for their animal's long-term well-being (Ioannidou et al., 2003).

Whereas the benefits from domain-oriented language pieces are evident from the example above, such a method is not always the most appropriate. The language can quickly get verbose and more importantly its customized components become incompatible with the language of the standard system.

5.5.2 *example: Explanations via Animated Speech and Animated tool tips*

Syntonicity can manifest itself not only as customized language pieces, but also in the form of explanations of the language components. In AgentSheets for example, individual commands and entire rules are syntonically explained via a unique combination of animation and speech (in the Mac version) or animation and textual tool tips (in the Windows version). When the "Explain" button is pressed when a single command is selected, the system steps through each component of the command annotating with blinking animation and verbally explains (either with speech synthesis or animated text in tool tips) what the command does. For instance, for the WWW read condition, the system explains that the condition is true in the context of an agent, if that agent finds the string specified when reading a specified Web page (Figure 18). First person is used to stress the fact that language pieces only make sense in the context of an agent, as pieces of that agent's behavior.

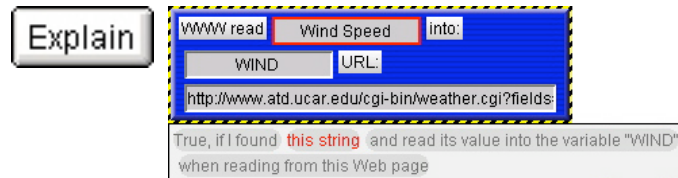


Figure 18. *An Animated Speech/Tooltip will explain command through a complete English sentence. Speech/Animation is synchronized with the selection of command parameters to establish the correspondence.*

Explanations are not static but they will interpret the current values of parameters. In some cases this will result in simple variations of sentences whereas in other cases the explanatory sentence will be more considerably restructured in order to clarify the meaning to the user (Figure 19).



Explanation: Remove me from the worksheet

Explanation: Erase the agent to my right.

Figure 19. *Explanation variations depending on parameters*

Explanations reduce challenges based on the comprehension of programs. At the same time they eliminate the need for languages to be more verbose which is often considered a good property for beginning programmers but gets in the way for more experienced programmers.

5.6 Allow Immersion

Immersing end-user programmers into the task and helping them experience the results of their programming activity by directly manipulating and interacting with their artifacts is an important factor for keeping them in the optimal flow part of the learning experience.

5.6.1 example: LEGOsheets and direct control of motors

LEGOsheets (Gindling, Ioannidou, Loh, Lokkebo, & Repenning, 1995) is a programming, simulation and manipulation environment created in AgentSheets for controlling the MIT Programmable Brick. The brick, developed at the MIT Media Lab as the research prototype of what is now known as LEGO Mindstorms, receives input from sensors, such as light and touch sensors and controls effectors, such as motors, lights and beepers. The combination of LEGOsheets and the Brick gives children the ability to create physical artifacts (vehicles and robots) and program

them with interesting behaviors (Resnick, 1994; Resnick, Martin, Sargent and Silverman, 1996).

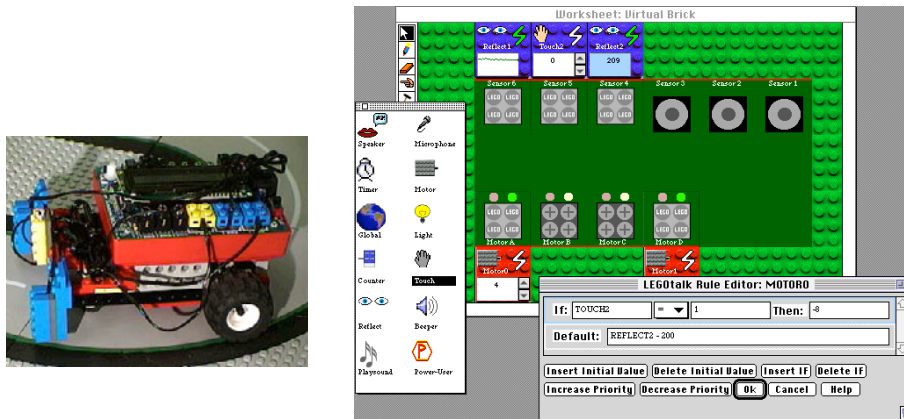


Figure 20. *LEGOsheets* a programming environment to program and directly control the MIT programmable brick. *LEGOsheets* programs are highly parallel putting rule-based behaviors into sensor effector agents.

A lot of the children's excitement and engagement with *LEGOsheets* arose from the physical aspect that the Brick provided. It is interesting to create a simulation of a car running around on a computer screen, but it is richer and more interesting when the car is programmed to do so in the real world. The richness of the resulting behavior of the behaving artifact did not come from the complexity of the program itself, but from its interactions with the real world (Simon, 1981).

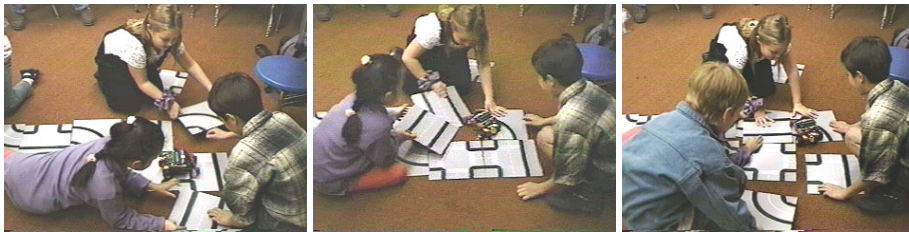


Figure 21. *"Programming" the Vehicle in the Real World*

Giving the opportunity to children to engage in interesting explorations in the world as part of social activities not only provided an engaging but also a highly motivating atmosphere that enabled even 3rd grade elementary school kids to take the step towards programming.

5.6.2 *example: Mr. Vetro, the simulated human being*

Mr. Vetro is an application we have developed using a unique architecture for compact, connected, continuous, customizable, and collective simulations (C5). This architecture can be generally geared towards helping students to experience and understand all kinds of complex distributed systems such as the human body, economies, ecologies, or political systems.

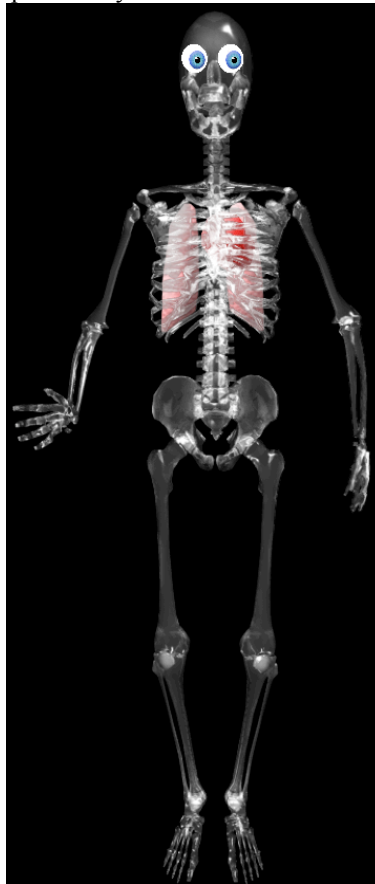


Figure 22. *Mr. Vetro is a distributed simulation of a human being.*

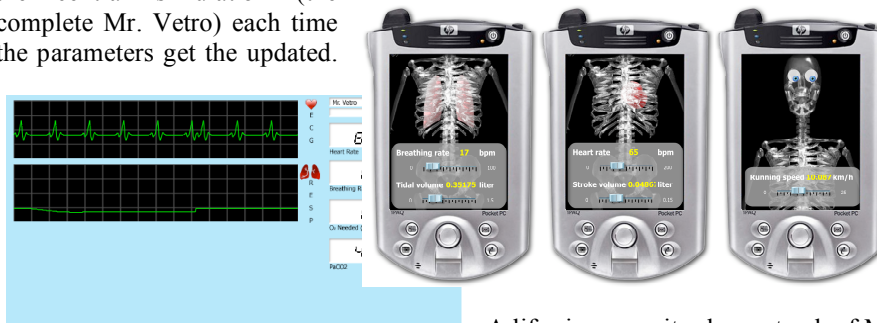
Mr. Vetro1 (left) is a simulated human being with a collection of simulated organs (such as heart and lungs) each of which are distributed as client simulations running on handhelds (right).

Using these client simulations users can control Mr. Vetro's organs. For instance, a group of students can control his lungs by varying parameters such as the breathing rate and tidal volume as a response to changing conditions such as

¹ Translated from Italian, "vetro" means "glass". The name is derived from Mr. Vetro's glass skeleton.

exercise or smoking. Another group can control Mr. Vetro's heart by varying heart parameters such as heart rate and stroke volume. A third group can act as the decision-making part of Mr. Vetro's brain to control decisions such as engaging in exercise and the intensity of the exercise.

With a wireless network, the client simulations send data to the server running the central simulation (the complete Mr. Vetro) each time the parameters get the updated.



A life signs monitor keeps track of Mr. Vetro's vital signs and displays them in the form of graphs or numerical values. O2 saturation in the blood, partial pressure of CO2, and O2 delivered to tissue are some of the values calculated and displayed.

Activities with Mr. Vetro are compelling and engaging as they promote interesting inter-group as well as intra-group discussions and collaborations to solve the tasks presented to students. Moreover they provide new ways to learn, not previously available by simply reading about human organs and systems in books.

Direct manipulation interfaces of changing the organ parameters allow users to change the simulation without having to engage in anything that can be perceived as traditional programming. As skills increase – mainly domain skills, but also skills related to interacting with the system – students can be exposed to more complex end-user development activities.

End-User Development related to Mr. Vetro takes place at two levels. At the highest level the handheld devices representing Mr. Vetro's organs have become the building blocks of a collaborative design activity. At the lower level users employ end-user programming to script organs or to analyze physiological variables. Teachers, for instance, may want students to express rules to determine different states of Mr. Vetro that need to be identified and addressed. Students could use an end-user language such as VAT to express a rule relating the level of partial pressure of CO2 to hyperventilation and hypoventilation.

[...] when ventilation is normal the partial pressure is about 40 mm Hg. Hyperventilation: means that Alveolar Ventilation is excessive for metabolic needs. The [partial pressure] PaCO₂ is less than 35 mm Hg. Hyperventilation may occur in response to hypoxia or anxiety. Hypoventilation means that the Alveolar Ventilation is too low for metabolic needs and that the PaCO₂ is more than 45 mm Hg. The most common cause of hypoventilation is respiratory failure (Berne & Levy, 2000).

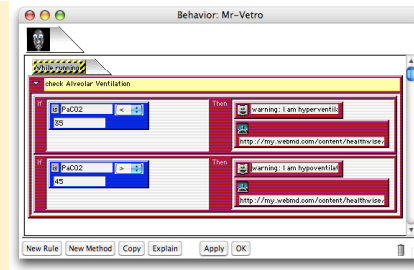


Figure 23. *The physiological rules can be directly turned into Visual AGenTalk. Rules are used to recognize physiological conditions and link them to existing Web information such as WebMD explaining condition and treatment.*

5.7 Scaffold Typical Designs

Whereas modeling is a desired computational literacy (diSessa, 2000) for a wide spectrum of computer end-users, programming is typically not. Therefore engaging in modeling activities should not necessarily have as a prerequisite the need to learn programming, especially in classroom settings. On the one hand, given the pragmatic concerns of heavy time limitations, using existing simulations is much easier and much more attainable in current educational settings than building simulations from scratch, even with EUD approaches. On the other hand, building simulations is an educationally effective activity (Ioannidou et al., 2003; Ioannidou, Repenning, & Zola, 1998; Technology, 1997; Wenglinsky, 1998; Zola & Ioannidou, 2000). Therefore, finding a middle ground would be essential for making simulations viable educational tools for mainstream classrooms. One such way would be to provide scaffolding mechanisms for the model-creation process. Scaffolding is the degree of structure provided by a system (Guzdial, 1994). High-level behavior specification paradigms provide a lower threshold to programming and therefore can be considered scaffolding mechanisms.

5.7.1 Example: Programmorphosis: Multi-layered programming with the Behavior Wizard

The Programmorphosis approach (Ioannidou, 2002, 2003) was developed as a multi-layered approach to end-user programming, which, at the highest level, enables novice end-user programmers to define behaviors of interacting agents in a high-level abstract language. In Programmorphosis, behavior genres are used to group and structure domain concepts in a template. Therefore, the task of programming is

elevated from a task of synthesis to one of modification and customization of existing behavior templates.

The Behavior Wizard was added to AgentSheets to instantiate Programmorphosis. Specifying behaviors is achieved by altering behavioral parameters in templates in a wizard environment that subsequently generates lower-level executable code. For instance, the behavior of an Octopus animal agent for an ecosystem simulation would be represented in Visual AgenTalk as shown in Figure 24 (left), with if-then rules defining eating, mating, and moving behaviors. In the Behavior Wizard, the user would specify the same behavior by manipulating parameters such as prey, hunting effectiveness, reproduction rate, as shown in Figure 24 (right).

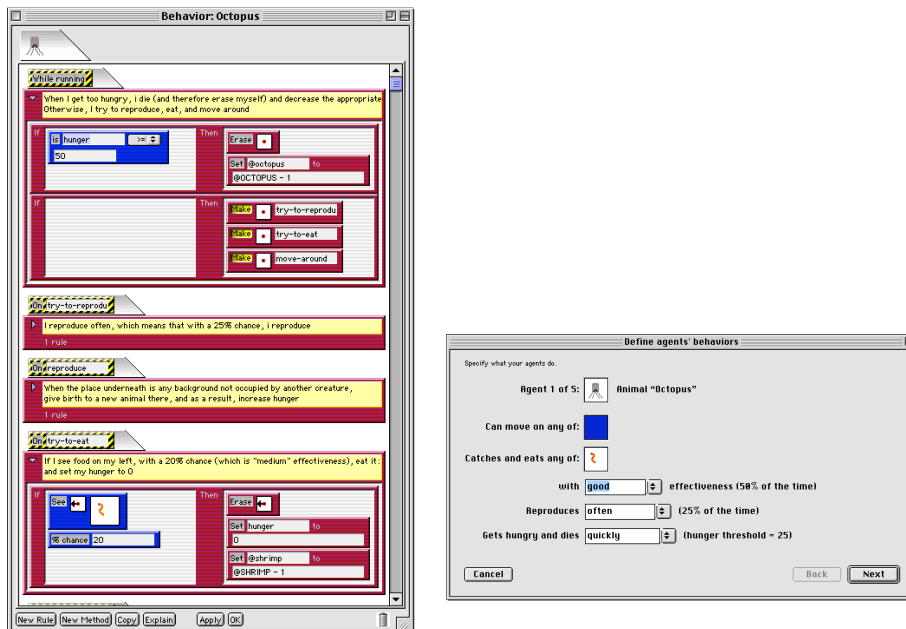


Figure 24. Behavior of an Octopus animal agent expressed in AgentSheets Visual AgenTalk (left). The same behavior expressed in the Behavior Wizard using the animal template (right)

The high-level behavior specification language featured with the Behavior Wizard essentially adds a layer in the programming process. As a result, this multi-layered programming approach enables a wide range of end-users to do the programming because multiple levels of abstraction address the different needs or levels of ability. At the higher level, a novice end-user may be “programming” declaratively through wizards, revealing meaningful customizations of existing

behaviors. At the lower level, a user may be programming procedurally in a programming language such as a rule-based language.

A general trade-off exists between the expressiveness of a programming language and its ease of use. Instead of selecting a fixed point in this space, Programmorphosis adds a high-level behavior specification layer and introduces a multi-layered approach to programming. Ideally, these programming layers should be connected to allow end-users to gradually advance, if they want to, from very casual end-user programming, which may be as simple as changing a system preference, to sophisticated end-user programming.

5.8 *Build Community Tools*

For end-users to harness the power of the Web and be encouraged for more active and productive participation, the image of the Web as a broadcast medium should be expanded to include end-user mechanisms that support collaborative design, construction and learning. This can be done by supporting:

- Bi-directional use of the web: Enable and motivate consumers of information to become producers of resources on the web.
- Richness of content: Make rich and expressive computational artifacts, such as simulation components and behaving agents, utilizing the web as a forum of exchange.

The Behavior Exchange is one such forum that achieves that.

5.8.1 *example: The Behavior Exchange*

The Behavior Exchange (Alexander Repenning & Ambach, 1997; Alexander Repenning, Ioannidou, & Phillips, 1999; Alexander Repenning, Ioannidou, Rausch, & Phillips, 1998), a Web-based repository that enables the sharing of simulation components, namely agents. The Behavior Exchange enables white-box reuse of agents by allowing inspection of agents acquired from the exchange as well as modification of their behavior because the full specification of agents' behaviors comes along with them when they are downloaded.

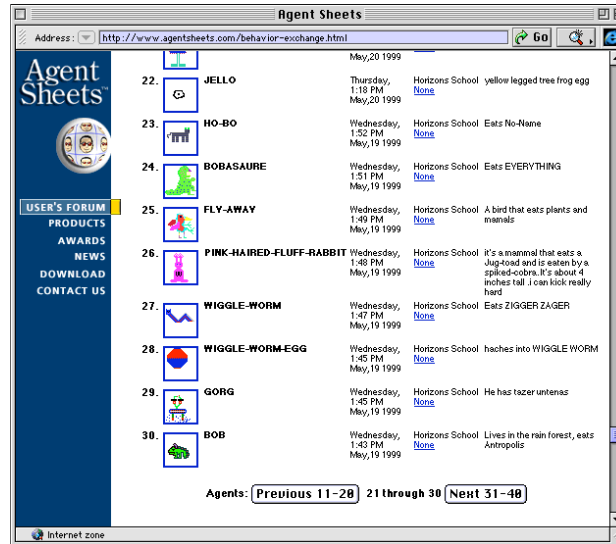


Figure 25. The AgentSheets Behavior Exchange. Users can give and take agent including descriptions what these agents do and who they look like.

The Behavior Exchange contains two kinds of information: informal and formal. Informal information is not interpreted by the computer. The look of an agent, textual descriptions concerning what the agent does, who created it, why and how it is used belong into the informal information category. Formal information is interpreted by the computer. All the rules that determine the behavior of an agent are considered formal information. The combination of informal and formal information turns these agents into a social currency of exchange. Users produce agents and share them. Other users pick them up and modify them to better fit into their own environment. This reuse mechanism allows a community of users to build and incrementally improve simulation content. The ability to build simulations by combining and modifying agents makes the agent level ideal for supporting collaboration among users, whether they reside in the same physical location or not, and for supporting the scaffolding (Guzdial, 1994) of the simulation creation process.

6. CONCLUSIONS

There cannot be one universal end-user development tool useful for all possible application contexts. Whether an end-user development is useful and gets accepted by an end-user community for a certain type of application depends on a number of

factors. The presentation formats used are of secondary relevance. A "useful and usable (Fischer, 1987, 1993) end-user development tool does not need to be iconic, visual, or textural for that matter. However, one perspective that we do think is universal is the viewpoint of end-user development as a learning experience balancing challenges and skills. A variety of scaffolding mechanisms presented in this article can help in making this learning experience more manageable. We have outlined a number of scaffolding mechanisms and extrapolated thirteen design guidelines from our experience with the AgentSheets simulation-authoring environment:

1. Make syntactic errors hard
2. Make syntactic errors impossible
3. Use Objects as Language Elements
4. Make domain-oriented languages for specific EUD
5. Introduce Meta-Domain orientation to deal with general EUD
6. Support Incremental Development
7. Facilitate Decomposable test units
8. Provide Multiple Views with Incremental disclosure
9. Integrate Development Tool with Web Services
10. Encourage Syntonicity
11. Allow Immersion
12. Scaffold Typical Designs
13. Build Community Tools

ACKNOWLEDGEMENTS

This work has been supported by the National Science Foundation (ITR 0205625, DMI 0233028).

7. REFERENCES

- Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A Pattern Language: Towns, Buildings, Construction*. New York, NY: Oxford University Press.
- Bell, B., & Lewis, C. (1993). ChemTrains: A Language for Creating Behaving Pictures. 1993 IEEE Workshop on Visual Languages, Bergen, Norway, 188-195.
- Berne, R. M., & Levy, M. N. (2000). *Principles of Physiology* (3rd ed.). St. Louis: Mosby.
- Blackwell, A. (1996). Metacognitive Theories of Visual Programming: What Do We Think We Are Doing? Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, 240-245.
- Brand, C., & Rader, C. (1996). How Does a Visual Simulation Program Support Students Creating Science Models? Proceedings of the 1996 IEEE Symposium of Visual Languages, Boulder, CO, 102-109.
- Brand, C., Rader, C., Carlone, H., & Lewis, C. (1998). Prospects and Challenges for Children Creating Science Models. National Association for Research in Science Teaching, San Diego, CA.
- Burnett, M. (1999). Visual Programming. In J. G. Webster (Ed.), *Encyclopedia of Electrical and Electronics Engineering*. New York: John Wiley & Sons Inc.
- Burnett, M., & Baker, M. (1994.). A Classification System for Visual Programming Languages. *Journal of Visual Languages and Computing*, 287-300.

- Cherry, G., Ioannidou, A., Rader, C., Brand, C., & Repenning, A. (1999). *Simulations for Lifelong Learning*. NECC, Atlantic City, NJ.
- Craig, B. (1997). *AGES: Agentsheets Genetic Evolutionary Simulations*. Unpublished Masters Thesis, University of Colorado, Boulder, CO.
- Csikszentmihalyi, M. (1990). *Flow: The Psychology of Optimal Experience*. New York: Harper Collins Publishers.
- Cypher, A. (1993). *Watch What I Do: Programming by Demonstration*. Cambridge, MA: The MIT Press.
- diSessa, A. (2000). *Changing Minds: Computers, Learning, and Literacy*. Cambridge, MA: The MIT Press.
- Eisenberg, M., & Fischer, G. (1994). *Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance*. Proceedings of the 1994 ACM CHI Conference, Boston, MA, 431-437.
- Fischer, G. (1987). *Making Computers more Useful and more Usable*. 2nd International Conference on Human-Computer Interaction, Honolulu, Hawaii.
- Fischer, G. (1993). *Beyond Human Computer Interaction: Designing Useful and Usable Computational Environments*. People and Computers VIII: Proceedings of the HCI'93 Conference, 17-31.
- Fischer, G., & Girgenson, A. (1990). *End-User Modifiability in Design Environments*. CHI '90, Conference on Human Factors in Computing Systems, Seattle, WA, 183-191.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley.
- Gindling, J., Ioannidou, A., Loh, J., Lokkebo, O., & Repenning, A. (1995). *LEGOsheets: A Rule-Based Programming, Simulation and Manipulation Environment for the LEGO Programmable Brick*. Proceeding of Visual Languages, Darmstadt, Germany, 172-179.
- Glinert, E. P. (1987). *Out of flatland: Towards 3-d visual programming*. IEEE 2nd Fall Joint Computer Conference, 292-299.
- Green, T. R. G. (1989). *Cognitive Dimensions of Notations*. Proceedings of the Fifth Conference of the British Computer Society, Nottingham, 443-460.
- Green, T. R. G., & Petre, M. (1996). *Usability Analysis of Visual Programming Environments: a 'cognitive dimensions' framework*. Journal of Visual Languages and Computing, 7(2), 131-174.
- Guzdial, M. (1994). *Software-Realized Scaffolding to Facilitate Programming for Science Learning*. Interactive Learning Environments, 4(1), 1-44.
- Ioannidou, A. (2002). *Programmorphism: Sustained Wizard Support for End-User Programming*. Unpublished Ph.D. Thesis, University of Colorado, Boulder.
- Ioannidou, A. (2003). *Programmorphism: a Knowledge-Based Approach to End-User Programming*. Interact 2003: Bringing the Bits together, Ninth IFIP TC13 International Conference on Human-Computer Interaction, Zürich, Switzerland.
- Ioannidou, A., Rader, C., Repenning, A., Lewis, C., & Cherry, G. (2003). *Making Constructionism Work in the Classroom*. International Journal of Computers for Mathematical Learning, 8, 63-108.
- Ioannidou, A., & Repenning, A. (1999). *End-User Programmable Simulations*. Dr. Dobb's(302 August), 40-48.
- Ioannidou, A., Repenning, A., & Zola, J. (1998). *Posterboards or Java Applets?* International Conference of the Learning Sciences 1998, Atlanta, GA, 152-159.
- Jones, C. (1995). *End-user programming*. IEEE Computer, 28(9), 68-70.
- Klann, M. (2003). *D1.1 Roadmap: End-User Development: Empowering people to flexibly employ advanced information and communication technology: EUD-Net: End-User Development Network of Excellence*.
- Lewis, C., Polson, P. G., Wharton, C., & Rieman, J. (1990). *Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces*. SIGCHI conference on Human Factors in Computing Systems: Empowering people, Seattle, Washington, USA, 235-242.
- Lewis, C., & Rieman, J. (1993). *Task-centered User Interface Design - A Practical Introduction*. Boulder, CO: A shareware book that can be downloaded from ftp.cs.colorado.edu/pub/cs/distribs/clewis/HCI-Design-Book/.
- Lieberman, H. (2001). *Your Wish Is My Command: Programming by Example*. San Francisco, CA: Morgan Kaufmann Publishers.
- Nardi, B. (1993). *A Small Matter of Programming*. Cambridge, MA: MIT Press.
- Norman, D. A. (1986). *Cognitive Engineering*. In *User Centered System Design* (pp. 31-61). Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers.

- Pane, J. F., & Myers, B. A. (1996). Usability Issues in the Design of Novice Programming Systems (Technical Report No. CMU-CS-96-132). Pittsburg, Pennsylvania: School of Computer Science, Carnegie Mellon University.
- Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. New York: Basic Books.
- Papert, S. (1993). *The Children's Machine*. New York: Basic Books.
- Papert, S., & Harel, I. (Eds.). (1993). *Constructionism*. Norwood, NJ: Ablex Publishing Corporation.
- Paternò, F. (2003). D1.2 Research Agenda: End-User Development: Empowering people to flexibly employ advanced information and communication technology: EUD-Net: End-User Development Network of Excellence.
- Rader, C., Brand, C., & Lewis, C. (1997). Degrees of Comprehension: Children's Understanding of a Visual Programming Environment. Proceedings of the 1997 Conference of Human Factors in Computing Systems, Atlanta, GA, 351-358.
- Rader, C., Cherry, G., Brand, C., Repenning, A., & Lewis, C. (1998). Principles to Scaffold Mixed Textual and Iconic End-User Programming Languages. Proceedings of the 1998 IEEE Symposium of Visual Languages, Nova Scotia, Canada, 187-194.
- Repenning, A. (1995). Bending the Rules: Steps toward Semantically Enriched Graphical Rewrite Rules. Proceedings of Visual Languages, Darmstadt, Germany, 226-233.
- Repenning, A., & Ambach, J. (1996a). Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing. Proceedings of the 1996 IEEE Symposium of Visual Languages, Boulder, CO, 102-109.
- Repenning, A., & Ambach, J. (1996b). Visual AgenTalk: Anatomy of a Low Threshold, High Ceiling End User Programming Environment. submitted to Proceedings of UIST.
- Repenning, A., & Ambach, J. (1997). The Agentsheets Behavior Exchange: Supporting Social Behavior Processing. CHI 97, Conference on Human Factors in Computing Systems, Extended Abstracts, Atlanta, Georgia, 26-27.
- Repenning, A., & Ioannidou, A. (1997). Behavior Processors: Layers between End-Users and Java Virtual Machines. Proceedings of the 1997 IEEE Symposium of Visual Languages, Capri, Italy, 402-409.
- Repenning, A., Ioannidou, A., & Ambach, J. (1998). Learn to Communicate and Communicate to Learn. *Journal of Interactive Media in Education (JIME)*, 98(7).
- Repenning, A., Ioannidou, A., & Phillips, J. (1999). Collaborative Use & Design of Interactive Simulations. Proceedings of Computer Supported Collaborative Learning Conference at Stanford (CSCL'99), 475-487.
- Repenning, A., Ioannidou, A., Rausch, M., & Phillips, J. (1998). Using Agents as a Currency of Exchange between End-Users. Proceedings of the WebNET 98 World Conference of the WWW, Internet, and Intranet, Orlando, FL, 762-767.
- Repenning, A., & Perrone, C. (2000). Programming by Analogous Examples. *Communications of the ACM*, 43(3), 90-97.
- Repenning, A., & Perrone-Smith, C. (2001). Programming by Analogous Examples. In H. Lieberman (Ed.), *Your Wish Is My Command: Programming by Example (Vol. 43, pp. 90-97)*: Morgan Kaufmann Publishers.
- Repenning, A., & Sumner, T. (1992). Using Agentsheets to Create a Voice Dialog Design Environment. Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing, Kansas City, MO, 1199-1207.
- Repenning, A., & Sumner, T. (1995). Agentsheets: A Medium for Creating Domain-Oriented Visual Languages. *IEEE Computer*, 28(3), 17-25.
- Schneider, K., & Repenning, A. (1995). Deceived by Ease of Use: Using Paradigmatic Applications to Build Visual Design. Proceedings of the 1995 Symposium on Designing Interactive Systems, Ann Arbor, MI, 177-188.
- Simon, H. A. (1981). *The Sciences of the Artificial* (second ed.). Cambridge, MA: MIT Press.
- Smith, D. C. (1975). PIGMALION: A Creative Programming Environment (Technical Report No. STAN-CS-75-499): Computer Science Department, Stanford University.
- Smith, D. C., Cypher, A., & Spohrer, J. (1994). KidSim: Programming Agents Without a Programming Language. *Communications of the ACM*, 37(7), 54-68.
- Technology, P. (1997). Report to the President on the Use of Technology to Strengthen K-12 Education in the United States

- Wenglinsky, H. (1998). Does it Compute? The Relationship Between Educational Technology and Student Achievement in Mathematics. Princeton, NJ: Educational Testing Service.
- Zola, J., & Ioannidou, A. (2000). Learning and Teaching with Interactive Simulations. *Social Education: the Official Journal of National Council for the Social Studies*, 64(3), 142-145.