Visual Language Features Supporting Human-Human and Human-Computer Communication

Jason E. Robbins[1], David J. Morley[2], David F. Redmiles[1], Vadim Filatov[3], Dima Kononov[3]
[1] University of California, Irvine [2] Rockwell International [3]RR-Gateway, AO

Abstract

Fundamental to the design of visual languages are the goals of facilitating communication between people and computers, and between people and other people. The Object Block Programming Environment (OBPE) is a visual design, programming, and simulation tool which emphasizes support for both human-human and human-computer communication. OBPE provides several features to support effective communication: (1) multiple, coordinated views and aspects, (2) customizable graphics, (3) the "machines with push-buttons" metaphor, and (4) the host/transient pattern. OBPE uses a diagram-based, visual object-oriented language that is intended for quickly designing and programming visual simulations of factories.

1. Introduction

Complex visual programs are difficult to construct, understand, and evolve. These difficulties are compounded when a group of people need to achieve a shared understanding of complex software to carry out a task. Additionally, designers need to express their mental model of a design in a formal model to benefit from computer simulation, analysis, and refinement. These two needs reflect two challenges to any visual language: human to human communication and human to computer communication.

We examine these challenges in the domain of designing factory automation software, although they are evident in software development regardless of domain. Rockwell International developed the *Object Block Programming Environment* (OBPE) design, programming, and simulation tool to address these challenges.

OBPE uses a diagram-based, visual object-oriented language [13] that is intended for quickly designing and programming visual simulations of factories. That domain has encouraged us to orient our thinking and designs very closely around real-world objects. When designing software we first look at existing real world objects and their relationships and then try to model them as closely as possible while abstracting details that distract attention from our main concerns.

Currently, OBPE exists as a research prototype serving to explore the concepts and features that are needed to allow effective visual programming in our domain. OBPE programs are composed of object blocks, ports, and arcs. *Object blocks* use *ports* as their interface points to the rest of the system. *Arcs* are message passageways between ports. Object blocks (or simply, blocks) are visual abstract data types [1] which encapsulate state, behavior, visualization of state and behavior, and user interface event processing. Furthermore, blocks are first class objects because they are instances of normal Smalltalk classes.[1]

Users interact with OBPE through browsers that allow direct manipulation of blocks and visualization of their state. Users can modify the visualized state directly, e.g., dragging on the hands of a clock face changes the time, and they can invoke operations by pushing buttons on the block itself. Multiple browsers can view the same block at the same time, and all views update automatically. OBPE has no notion of run time or compile time: any block may be manipulated at any time, and the internal logic of any block can be modified at any time.

The main thrust of this paper is to discuss the features of OBPE that support human-human and human-computer communication. Section 2 provides a conceptual framework for the communication needs in program development. Section 3 provides an overview of OBPE. Section 4 discusses specific features of OBPE that address the needs raised in Section 2. Finally, OBPE is placed in the context of related work.

2. Problem statement

One of our goals in building OBPE is to provide support for the various groups of people who are involved in developing and using factory automation software. The needs of these various stakeholders impose diverse requirements and constraints. Factory designers want to model the factory and its software in a way that fits their mental model of

This research is supported in part by the Air Force Material Command and the Advanced Research Projects Agency under Contract Number F30602-94-C-0218. Additional support is provided by Rockwell International. The content of the information does not necessarily reflect the position or the policy of the funders and no official endorsement should be inferred.

1. This paper presents OBPE as an extension to Smalltalk, although versions of OBPE have been implemented in C++, and Objective-C.

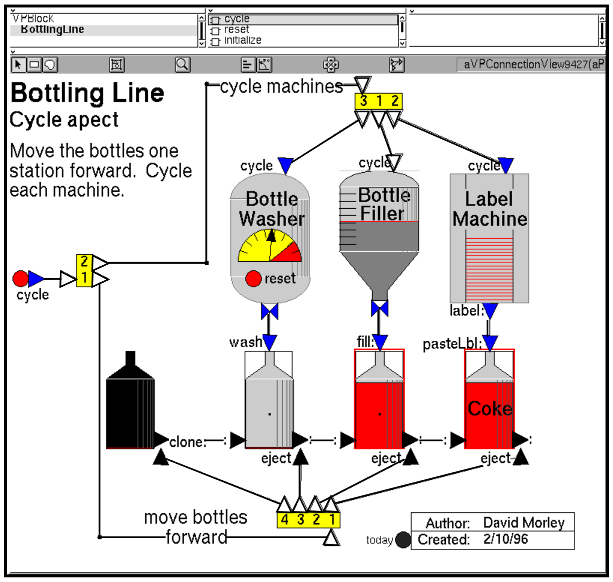
the factory. Designers should be able to communicate their designs to corporate decision makers in a form that is clear and persuasive. Staff programmers should be able to use a model both as an active specification and in implementing a working control system.[2] Machine operators on the factory floor need to understand the correspondence between the real machines and the model, not only in situations that the model is designed for, but more importantly in exceptional situations. Designers must communicate their mental model of the factory to the operators so that they may relate it to their own.

These issues are the same that are faced generally in software design. The need for human-human communication is evident in both team software development and in the designer's task of communicating with decision makers and users. The need for human-computer communication is evident in the tasks of writing programs, and interpreting program output. These two needs become more related as software becomes more complex. In fact, the process of developing complex software can best be understood as a communication and learning process [7]. Much of the VL literature tacitly considers both types of communication; we do so explicitly.

Necessarily, design and programming languages allow humans to externalize their mental model of a given problem and solution. Once a designer's mental model is in an explicit form it can be simulated, analyzed, and communicated to other people. Thus, generally stated, the problem that we address in this paper is designing a visual language that supports human-human and human-computer communication.

3. Overview of OBPE and usage scenario

Figure 1 shows some blocks that model the machines found in a bottling plant: a bottle washer, a filler, and a labeler. On each block are buttons that operate the block, such as **cycle** which performs all the steps of the machine's main task and then returns to a known, safe state. For example, the **cycle** button on the washer causes a sequence of varying water pressures and temperatures in the attached Bottle. The **cycle** button is actually an input port, as will be further discussed in Section 4.3. The smaller blocks in Figure 1 perform sequencing: when a message is sent to the input port of the top sequence block (labeled "312"), it sends one message via each output port in order. The diagram in Figure 1 defines one *aspect* of class BottlingLine. In fact, all diagrams exist within some *block class*, just as all Smalltalk methods exist within some class.



Bottling Line
Cycle apect
Move the bottles one station forward. Cycle each machine.

Figure 1. Class BottlingLine, aspect cycle

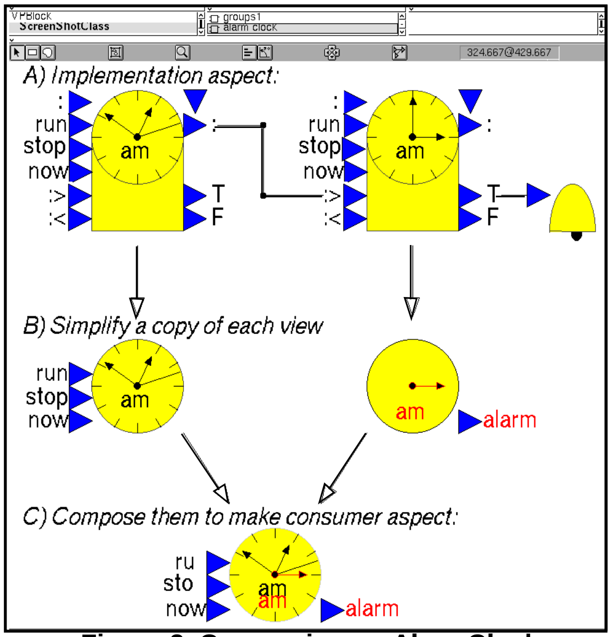


Figure 2. Composing an AlarmClock

Figure 2a shows how an alarm clock can be composed of three components: two simple clocks and a bell. The left-hand clock stores the current time, the right-hand clock stores the alarm time. Whenever the left-hand clock is running it broadcasts its current time.[3] The right-hand clock compares that time against its own. When the alarm time is reached the bell starts to ring.

2. An eventual goal of OBPE is to directly control factory devices. Some experiments have been carried out toward that goal, but OBPE can also support a prototype-then-reimplement development process.

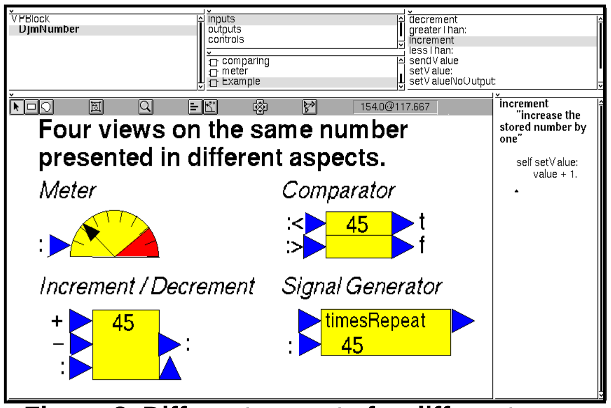
3. OBPE uses concurrency so that active blocks like the clock have their own thread of control. This also allows the user to edit a diagram while it is running. Any diagram with active blocks is virtually always running.

Input ports are parts of both the user interface and programmatic interface of a block. Input ports on a block are associated with Smalltalk methods defined in that block's class. *Output ports* are associated with output methods that package and send arguments out of the block. In sending a message from one block to another the following events occur: (1) the sending block invokes its own output method, (2) that method packages its arguments and sends them over any attached arcs, and (3) any receiving input ports use their own message selectors to invoke methods on the receiving blocks. The sending block does not specify which other blocks receive the message, nor what message selector to use; that information is determined by the connectivity of blocks. This type of implicit invocation has been used in a variety of systems to facilitate late binding and reuse [15, 22]. Furthermore, input and output ports together delimit the network of objects that make up a block, thus bounding recursive operations such as deepCopy [12].

Some diagrammatic conventions are evident in the figures: input ports have triangular shapes that point toward the body of the block or upward, output ports have triangular shapes that point away, ports are labeled with the name of the operation that they perform, colons indicate arguments, and ports labeled with only a colon broadcast or set the value of some obvious state variable. However, these are only conventions, not syntax rules, OBPE has a remarkably simple syntax. In fact, the desire to allow users to evolve their notation conflicts with complex, enforced syntactic rules. This conflict will be discussed briefly in Section 4.2.

Each block is an instance of a block class. Rather than construct new instances through code, OBPE takes a prototype-based approach [23]: each class stores a prototypical instance, and any instance can be cloned to make a new one (e.g., the leftmost Bottle in Figure 1). The prototypical instance of a class can be modified in the OBPE class browser (Figure 3 and all other screen shots). Block classes are simply Smalltalk classes and thus encapsulate state and behavior. Block classes form a class hierarchy and inherit code and instance variables as one would expect; they also inherit aspects, ports, and subcomponent blocks. The OBPE class browser is identical to the Smalltalk class hierarchy browser, with the addition of panes for selecting and defining aspects. Individual panes in the OBPE browser may be resized or eliminated to make more room for others.

OBPE exhibits four characteristics common to VL's: conceptual simplicity, concreteness, explicitness, and liveness [1]. OBPE uses a small number of concepts, each of which is powerful and simple. Direct manipulation of blocks and diagrams makes them concrete and their relationships explicit. Multiple views and aspects maintain a high level of liveness. These characteristics cut across the features discussed in the next section.



```
increment
    "increase the stored number by one"

    self setValue:
        value + 1.
```

Figure 3. Different aspects for different uses

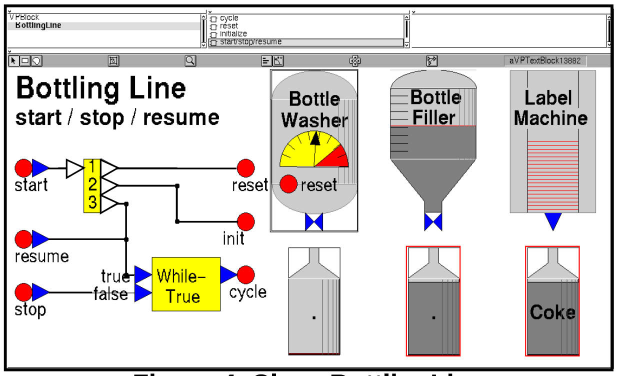


Figure 4. Class BottlingLine, aspect start/stop/resume

4. OBPE support for human-human and human-computer communication

OBPE meets the objectives described in Section 2 with a small number of powerful features and metaphors. The underlying theme of these features is correspondence of OBPE models with the mental models of various stakeholders involved in designing and using the system. Most VL designers implicitly consider communication of mental models, considering it explicitly yields more thorough support, as discussed in this section and Related work.

4.1 Multiple, overlapping aspects and views

OBPE allows each block to have multiple aspects, and different presentations associated with it. This allows a given block to appear in a way that is appropriate for a given usage (see Figure 3). Each aspect presents a subset of the ports defined on the block.

The use of multiple aspects in a single block separates concerns and factors complexity. The bottling line example shown in Figures 1 and 4 illustrates two aspects of class BottlingLine. Figure 1 shows logic related to a single cycle of the line, Figure 4 shows the logic for continuous processing. Each of these aspects is used to implement part of the functionality of the bottling line. Figure 5 shows the logic needed to reset the line to a known, safe state. The aspects contain distinct, coordinated views on the same three machines, e.g., operations that affect the filler are shown in all views. Because each view may present the block differently, each view may show the operation differently.

Separation of concerns implies that different concerns may be addressed by different people doing different tasks. *Using multiple aspects allows the same model to be viewed in different ways to support different people and tasks* [13]. Figure 6 shows the operator's user interface to the bottling line. It presents the bottling line status and allows some predefined manipulations without showing any logic. The buttons in the operator aspect are simplified views on the same buttons depicted in the other aspects.

Abstraction mechanisms in many VL's [18, 5] and textual languages, e.g., Ada, separate a module's interface from its implementation. Those languages provide for a single programmatic interface while dividing the implementation into several methods and cases. OBPE supports this traditional abstraction mechanism. It also provides for more flexible abstraction via multiple, overlapping aspects. Figure 7 compares the traditional abstraction mechanism to the use of multiple aspects in OBPE. Heavy boxes represent the total interface and implementation of the module or block, each point represents a language statement or element. Each aspect contains part of the complexity, and may overlap other aspects (as seen in the bottling line). In fact, two aspects may overlap totally if they differ only in presentation. A single aspect may mix interface and implementation for convenience in early development, to make the correspondence between interface and logic explicit (as in Figure 4), or to demonstrate the operation of a block via "a black box with a window." Some of the aspects of a block may be considered public or private, depending on the amount and kind of interface contained. Overlapping aspects aid stakeholders in extending partial understanding of the system because unfamiliar blocks are shown in relation to familiar ones [13]. For example, a stakeholder familiar with the behavior of Figure 6 can leverage that knowledge to understand Figure 4, and from there to understand Figures 1 and 5. On each step, familiar blocks are related to (and in this example, surround) unfamiliar ones.

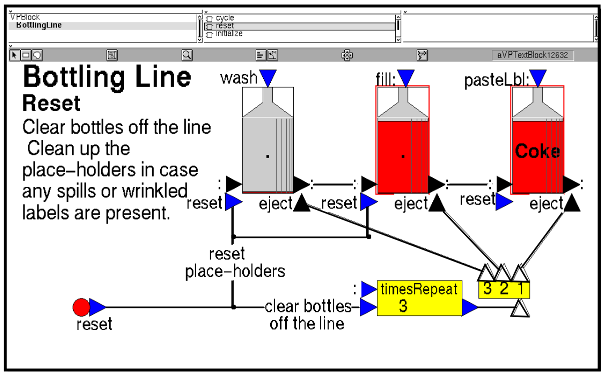


Figure 5. Class BottlingLine, aspect reset

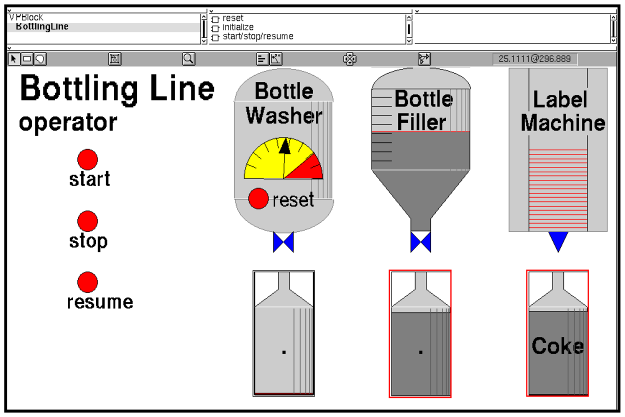


Figure 6. Class BottlingLine, aspect operator

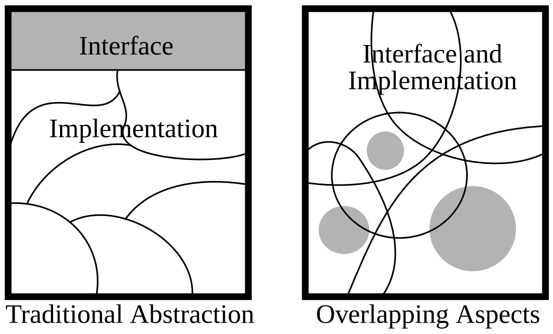


Figure 7. Two abstraction mechanisms

Decomposition of complexity can be hierarchical or alternative. In traditional top-down design the complexity of the bottling line would be hierarchically decomposed into various subcomponents, one of which is the washer. Strict hierarchical decomposition is well suited for structuring large amounts of complexity in one way, supporting one understanding of that complexity, and one task. Alternative decompositions are well suited for structuring complexity in complementary ways. The combination of decomposition techniques in OBPE allows designers to address multiple concerns in depth, while facilitating understanding by others.

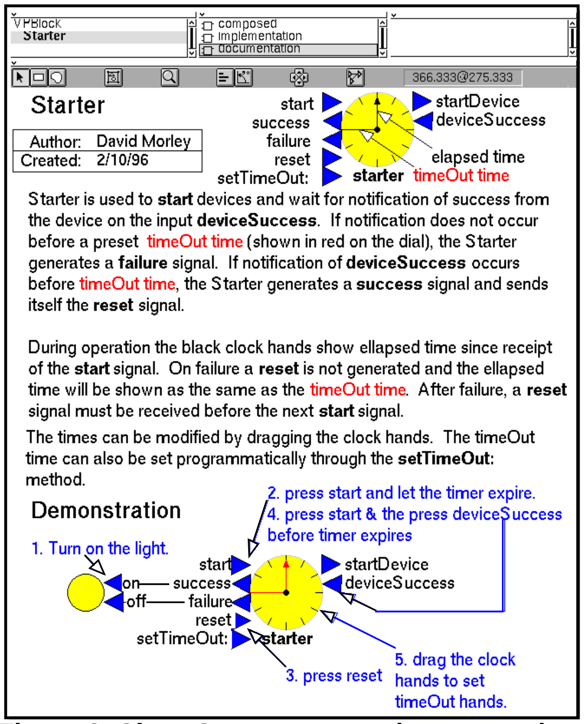
Using multiple aspects support correspondence between OBPE models and mental models by dividing complexity and separating concerns. Overlap among aspects provides context for stakeholders in understanding those aspects. Alternative decompositions support the different mental models of different stakeholders.

4.2 Customizable presentation graphics

During software development designers must convincingly communicate their designs to decision makers in a form that emphasizes issues over details of language syntax. The characteristics of the model being demonstrated must be tied to decision issues and arguments. Furthermore, the style of presentation must fit the social and professional norms of that group: it must look like a presentation, not a programmer's whiteboard.

OBPE empowers designers to make their work clear, concrete, informative, and persuasive. The editors used to construct diagrams have all the power and flexibility of commercial drawing and presentation packages. Diagrams may be annotated with unstructured graphical elements, such as stylized text and bitmaps, for documentation, argumentation, and notes (see Figure 8 and others). Informal graphical annotations that are used repeatedly may be formalized by defining a block with the same appearance (e.g., the authorship block in Figures 1 and 8). We view this capability in the context of a theory of incremental formalism [19]. Like pseudo-code, it has the benefit of allowing designers to get their ideas down easily without formalizing too early.

We trade syntax for convention. Two constructs make up the majority of OBPE syntax: the block-port-arc graph representation, and the constraint that output ports can only be connected to input ports with the same number of arguments. Any graphical element may be used as a port, not just triangles and circles. Ports need not be labeled according to our conventions, or labeled at all (e.g., one output port on the filler looks like a valve symbol). Leaving appearances to the designer means that they cannot be depended on for syntactic rules. As language designers we



Starter

Author:	David Morley
Created:	2/10/96

Starter is used to **start** devices and wait for notification of success from the device on the input **deviceSuccess**. If notification does not occur before a preset timeOut time (shown in red on the dial), the Starter generates a **failure** signal. If notification of **deviceSuccess** occurs before timeOut time, the Starter generates a **success** signal and sends itself the **reset** signal.

During operation the black clock hands show ellapsed time since receipt of the **start** signal. On failure a **reset** is not generated and the ellapsed time will be shown as the same as the timeOut time. After failure, a **reset** signal must be received before the next **start** signal.

The times can be modified by dragging the clock hands. The timeOut time can also be set programmatically through the **setTimeOut:** method.

Demonstration

1. Turn on the light.
2. press start and let the timer expire.
3. press reset
4. press start & the press deviceSuccess before timer expires
5. drag the clock hands to set timeOut hands.

Figure 8. Class Starter, aspect documentation

have given up the power to make a rule that, for example, red ports are synchronous and blue ports are asynchronous, although that can be a convention. We reinforce our conventions with extensive, reusable examples. Of course, we could extend the environment to warn the user when conventions are violated, as we have done in the Argo design environment [17]. Using OBPE in another domain would demand a new set of conventions, but those can be defined within the environment itself.

Each block instance stores a local, editable copy of each aspect. This allows blocks to have customized appearances and ports that make them better suited for each usage. This can reduce visual clutter, e.g., the blocks in Figure 1 show only a subset of their input ports. It can also facilitate reuse and composition of visualizations. Figure 2 shows three steps in the construction of an alarm clock: (a) two clocks and a bell are connected as described earlier, (b) new views on the clocks are created and each is edited to remove undesired graphical elements, (c) the two views are moved on top of each other to present a single clock face with an alarm hand. The event handlers of the two clocks are also composed, so that the user may drag on the normal hands to set the time and the alarm hand to set the alarm time. Reusing presentations and event handling supports scaling up to more complex blocks [2]. In order for reuse to take place, it must be easier to reuse components than to reinvent them [11]. In general, part of reuse is adaptation of the reused component to fit the new context. Customization in OBPE has the side benefit of easing reuse.

The main benefit of customizable presentation graphics is in human-human communication. Designers can outline their mental models in OBPE very quickly, without being forced to fit a structure predefined by the tool. The model can be presented to optimize recognition by humans, not parsing by computers.

4.3 The "machines with push-buttons" metaphor

Real factory machines have a variety of operator controls. The most common of these controls is the simple push-button that invokes a specific operation. A **cycle** button causes the machine to do its main sequence of actions. Other buttons are typically labeled with the names of individual steps in the machine's main sequence. They are provided to allow experimentation, testing, and recovery from exceptional situations. These same activities are central to the process of factory automation design.

In OBPE, blocks have the same buttons as the corresponding real machines. Users can push buttons to operate blocks just as they would real machines. The results of these operations are immediately visible. Direct manipulation of machines is so understandable, concrete, and compelling that we based our programming facility on it, i.e., *the user and programmatic interfaces of a block are the* *same*. OBPE blocks interoperate with other components by “pushing their buttons.” The sending of a message from one block to another can be thought of as one machine reaching out and pushing a button on another. The similarity of manipulation and programming is reinforced by using the same animation techniques for both. When the user clicks on an input port, it highlights. When one block sends a message to another block, the same highlighting is seen.

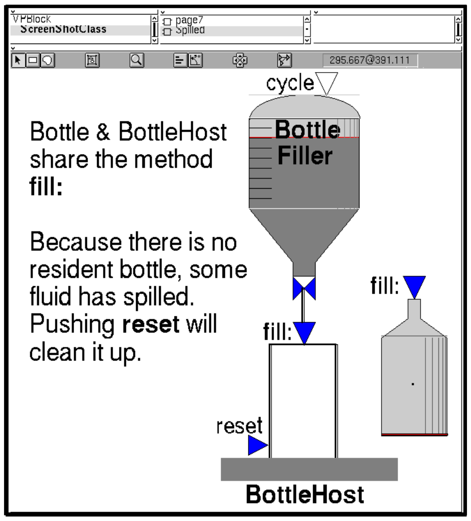
Of course, most real machines do not have buttons to invoke all the low level operations that are desirable in automation, but we need not abandon our metaphor, because we can imagine opening a panel to discover “power user” buttons that extend both the user and programmatic interfaces simultaneously. Obviously, a simple click does not supply arguments to a method; however, we tend toward operations that rely primarily on the state of the receiving block instead of arguments. In cases where arguments must be supplied, the user is prompted to supply them.

Interspersing programming and direct manipulation without switching mental contexts allows users of OBPE to test, debug, and understand programs by experimentation. This form of human-computer communication allows designers to effectively transform their mental models into programs, and understand simulation results. Human-human communication is bolstered when stakeholders demonstrate simulations to one another.

4.4 The host/transient pattern

We encountered the need for dynamic relationships in the factory domain when we attempted to model material flow, e.g., the motion of bottles among machines. The concept of material flow divides the designer’s mental model into two parts: *transient objects* that flow, and *stable objects* that do not. Each transient follows one of several possible trajectories, which themselves are stable.

In Figures 1 and 4 each machine is statically connected to a BottleHost block (presented as a hollow rectangle). When a Bottle is placed in a BottleHost (either by dragging it there, or programmatically) a dynamic relationship is formed between them, and messages sent to ports on the BottleHost are delegated to the Bottle if possible. If the Bottle is resident, it handles the **fill:** message, but the **eject** message is handled by the BottleHost because the Bottle does not implement **eject**. If the Bottle is removed (by dragging it out, or by invoking **eject**) the BottleHost handles all messages itself. The message selector is determined by the port on the host that receives the message. The Bottle implements **fill:** by raising its fluid level, the BottleHost models empty space by spilling fluid on the floor in response to **fill:** (Figure 9). Polymorphism allows other blocks, such as Jar or Can, to be resident in the BottleHost, and a Bottle block may be resident in different types of


Bottle & BottleHost share the method **fill:**

Because there is no resident bottle, some fluid has spilled. Pushing **reset** will clean it up.

Figure 9. A Host without a Transient

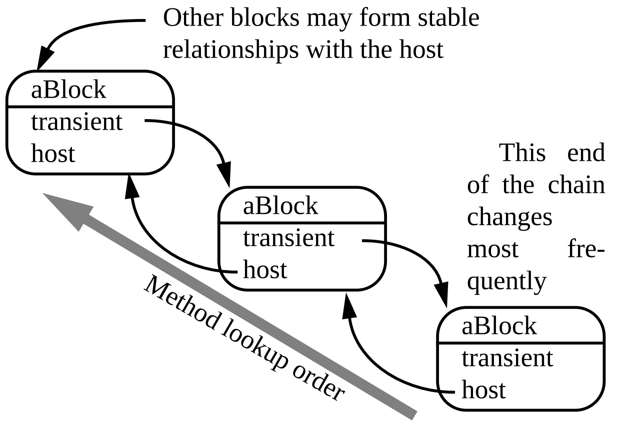

Other blocks may form stable relationships with the host
aBlock
transient
host
aBlock
transient
host
This end of the chain changes most frequently
aBlock
transient
host
Method lookup order

Figure 10. The host/transient pattern

hosts, e.g., Crate. In each case OBPE ensures that the appropriate method is activated.

OBPE supports controlled dynamism via a novel design pattern we call *host/transient* (Figure 10). *Each block may act as a host for a resident transient block to which it delegates as many messages as possible*. Blocks acting as transients may act as hosts for other blocks, e.g., dust inside a bottle inside a washer.

Modeling material flow via dynamic inheritance [23] or the Chain of Responsibility pattern [9] would allow the transient object to temporarily take on some features of the more stable object, thus clients must refer to the transient object. In contrast, the host/transient pattern allows the stable object to temporarily take on the features of the transient object. Thus, clients may simply refer to the stable object. This difference is key because it leverages the static nature of the diagram to structure dynamism and make it understandable.

Although the host/transient pattern was initially conceived to model material flow, applications to other domains are possible: patches and viruses can be modeled as transients resident in software hosts, and nested user interface modes can be modeled as both hosts and transients.

OBPE uses a diagram-based visual language in which connected graphs represent programs structured after mental models of the real world. Material flow is a natural and eminently understandable use of dynamism in the factory domain. The host part of host/transient makes it easier for designers to understand certain dynamic behaviors by concretely presenting the possible trajectories of transients. The host blocks provide user interface affordances [13] for designers to recognize and manipulate trajectories. The transient part of host/transient makes clearer the correspondence between the OBPE model and the designer's mental model by introducing the concept of transience. Without the concept of transience, dynamic relationships are only implicitly differentiated from static ones: every relationship appears equally likely to change, and dynamism can undermine users' confidence in their understanding of all parts of the model.

5. Related work

Function block diagrams are based on data flow diagrams and are commonly used in factory modeling. From our point of view, the function block paradigm is wrong for visual programming. Functions are verbs, and it is difficult to present verbs (except as the animation of nouns). Objects are nouns, and it is often easy and natural to present nouns. The superficial resemblance of function and object block diagrams is deceptive; indeed, the diagrams are the graph duals of each other. In data flow diagrams, nodes represent behavior and arcs represent state. In object block diagrams, nodes represent state and arcs represent behavior. This shift in focus is fundamental to the object-oriented paradigm.

Agentsheets [16] and KIDSIM [20] are two visual programming environments that emphasize goals similar to our own: ease of expressing mental models of problem solutions through object-orientation and customizable graphics. However, both limit the range of applications they can support due to a fixed grid representation and time-step (as opposed to discrete-event) simulation. The rule-based paradigm supports dynamism, but lacks clear affordances, as discussed in Section 4.4. While Agentsheets supports designers and end users with multiple editors, it does not allow definition of multiple, overlapping aspects for different stakeholders.

Vista [18] is a VL which has much in common with OBPE. Both are diagram-based languages with multiple views, nested components, and support for dynamic relationships. Aliases in Vista support an asymmetric form of multiple views in which one view is considered the original. Support for multiple aspects in Vista is limited to interface (iconic) or implementation (expanded), where implementation networks partition the logic of a module, as shown in Figure 7. Vista's notation is task-specific only in the amount of detail shown and whether text or graphics are used; whereas, OBPE provides for distinct aspects using notations evolved to satisfy different stakeholders. Both systems allow components to be composed, but only OBPE supports composition of visualizations and event handlers. Blocks are more concrete than Vista processors because of their liveness and the "machines with push-buttons" metaphor. Dynamic substitution of public subcomponents in Vista provides controlled dynamism, but it does not leverage default behaviors as the host/transient pattern does via delegation. Vista focuses on software engineering principles and human-computer communication; whereas, OBPE focuses on human-human and human-computer communication.

Instance-based and prototype based languages, such as Self [23], have influenced OBPE. We take a hybrid approach that uses both classes and prototypes: classes define some behavior via methods and state via instance variables, but other behaviors are encoded in aspects of prototypical blocks via the connectivity and local state of subcomponent blocks. Dynamic inheritance in Self allows arbitrary dynamism, whereas we desired controlled dynamism that reinforces correspondence with mental models.

The distinction between view-centered and object-centered environments, discussed in [6], requires each object be presented in only one location at a time, thus obeying a basic physical law. OBPE departs from the definition of object-centered by allowing multiple views on the same block that may appear very differently. We find that supporting liveness [21] is enough for users to understand that they are seeing multiple views of the same object and to feel as though they are manipulating it directly. Directness of multiple views is reinforced by the fact that when one view of a block is selected it is enclosed in a red (selection) highlight box, at the same time all other views of that same block are enclosed in blue (informative) highlight boxes.

Many authors have used the "software IC" metaphor: software components are like integrated circuits that combine to make larger components [4]. The difficulty in building on this metaphor is that integrated circuits have no user interface. Real IC's do nothing when held in one's hand, human beings are incapable of manipulating or querying an IC directly. In the software IC metaphor the user interface and programmatic interface are not bound together.

We have discussed issues and features that support human-human and human-computer communication. In so doing, we have addressed some of the needs designers face in expressing their mental models of problem solutions. However we have not discussed problem-domain specific support as suggested by many researchers including [8].

6. Conclusions

We have presented four features of OBPE that support human-human and human-computer communication by flexibly supporting the mental models of various stakeholders. Multiple, overlapping views and aspects help manage complexity, separate concerns, and facilitate reuse. Customizable graphics give designers the flexibility to evolve notation appropriate for the current domain, organization, project, task, and stakeholders. The push-button metaphor keeps factory models recognizably close to the real world, and encourages experimentation by uniting programmatic interfaces with user interfaces. The host/transient pattern provides affordances for controlled dynamism while reinforcing correspondence with mental models.

OBPE's simplicity and communication features are applicable to other domains. We have built two systems apply OBPE's block-port-arc representation, multiple aspects, and customizable graphics. In the software architecture domain, Argo is a design environment for Chiron-2 style GUI architectures [17, 22]. In the process modeling domain, the Rockwell Graphical Enterprise Modeler (RGEM) is a tool in experimental use at Rocketdyne. Additional experimental systems have modeled factory communication networks and extended the VisualWorks[4] GUI screen generator to graphically represent widget relationships.

7. Acknowledgments

We would like to acknowledge the many staff members at RR-Gateway, AO, in Moscow Russia who contributed countless ideas, not to mention hours, to the concepts and implementation of OBPE through its several versions. We also thank the management at Rockwell International Science Center which has so patiently supported this research.

8. References

[1] M. M. Burnett, "Seven Programming Language Issues", in [3].

[2] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, S. Yang, P. van Zee, "Scaling Up Visual Programming Languages," *IEEE Computer*, March 1995, pp. 45-54.

[3] M. M. Burnett, A. Goldberg, T. G. Lewis (Eds.) *Visual Object-Oriented Programming: Concepts and Environments*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[4] B. J. Cox. *Object Oriented Programming: An Evolutionary Approach*.

[5] P. T. Cox, F. R. Giles, T. Peitrzykowski, "Prograph," in [3]

[6] B. Chang, D. Ungar, and R. B. Smith. "Getting Close to Objects", in [3].

[7] Curtis, Krasner, and Iscoe. "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM*, 31(11):1268-1287, Nov. 1988.

[8] G. Fischer and A. C. Lemke, "Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication," *Human-Computer Interaction*, Vol. 3, No. 3, 1988, pp. 179-222.

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley. 1994.

[10] A. Goldberg, M. M. Burnett, T. G. Lewis, "What is Visual Object-Oriented Programming?" in [3].

[11] C. W. Krueger, "Software Reuse", *Computing Surveys*, 24(2):131-184, June 1992.

[12] D. Morley, S. Chiu, J. Robbins, G. Veolker, and T. Maddux. "Reusable Objects" *Technology of Object-Oriented Languages and Systems*, March 1991, CNIT Paris, France.

[13] D. A. Norman, *The Psychology of Everyday Things*. Basic Books, Inc. 1988.

[14] C. Rathke and D. F. Redmiles, "Improving the Explanatory Power of Examples by a Multiple Perspectives Representation," *Proceedings of the 1994 East-West Conference on Computer Technologies in Education* (EW-ED'94, Crimea, Ukraine), P. Busilovsky, S. Dikareva, J. Greer, V. Petrushin (Eds.), September 1994, pp. 195-200.

[15] S. P. Reiss, "Connecting Tools using Message passing in the Field Environment," *IEEE Software*, 7(4):57-66, July 1990.

[16] A. Repenning, T. and Sumner, "Agentsheets: A Medium for Creating Domain-Oriented Visual Languages," *Computer*, Vol. 28, No. 3., March 1995, pp.17-25.

[17] J. E. Robbins and D. F. Redmiles, "Software Design From the Perspective of Human Cognitive Needs," *Proceedings of the 1996 California Software Symposium* (Los Angeles, CA), April 1996.

[18] S. Schifer and J. H. Frohlich. "Visual Programming and Software Engineering with Vista", in [3].

[19] F. Shipman and R. McCall, "Supporting Knowledge-Base Evolution with Incremental Formalization," *Human Factors in Computing Systems, CHI'94 Conference Proceedings*, Boston, MA, 1994, pp. 285-291.

[20] D. C. Smith, A. Cypher, J. Spohrer, "KIDSIM: Programming Agents Without a Programming Language," *Communications of the ACM*, Vol. 37, No. 7, July 1994, pp. 55-67.

[21] S. L. Tanimoto, "Towards a Theory of Progressive Operators for Life Visual Programming Environments," *Proceedings of the 1990 IEEE Computer Society Workshop on Visual Languages*, Skokie, IL, 1990, pp. 80-85.

[22] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, D. L. Dubrow. "A Component and Message-based architectural style for GUI Software," *IEEE Transactions on Software Engineering*, to appear in 1996.

[23] D. Ungar and R. B. Smith, "Self: The Power of Simplicity," in *OOPSLA'87 Conference Proceedings*, published as *SIGPLAN Notices*, Vol. 22, No. 2. pp. 227-241.

4. VisualWorks is a trademark of ParcPlace-Digitalk.