To appear in Visual Object-Oriented Programming: Concepts and Environments, Margaret Burnett, Adele Goldberg, Ted Lewis, editors. 1994.

The Design of a Completely Visual Object-Oriented Programming Language

Wayne Citrin
Michael Doherty
Benjamin Zorn
Department of Computer Science
University of Colorado at Boulder

Abstract

Object-oriented languages provide powerful programming features such as polymorphism, inheritance, and dynamic dispatch. While these features allow complex programs to be written more easily, they also make debugging and understanding these programs more difficult. Object-oriented languages have relied on simple visualization tools such as class browsers to aid programmers in understanding their programs. In this paper, we argue that a completely visual object-oriented programming language, VIPR, has significant advantages over textual object-oriented languages. We describe how VIPR represents all aspects of object-oriented programs including objects, classes, inheritance, polymorphism, and dynamic dispatch. By completely visual, we mean that the semantics of programs written in our language can be entirely described by simple graphical rules. VIPR provides a framework for and integrates existing methods of understanding the structure and execution of visual programs. Also, we discuss the programming environment support required by VIPR and argue why the language, with this environment, will be usable by expert programmers for solving large problems.

1 Introduction

Object-oriented programming requires understanding a number of complex concepts, which can be difficult even for experienced programmers of procedural languages such as C and Pascal. In this paper, we describe a visual object-oriented language that has many of the object-oriented features of a language like C++, but whose semantics can be explained using simple graphical rules. We intentionally chose to provide semantics similar to C++ for two reasons. First, C++ provides the most important features of the object-oriented paradigm very efficiently and thus VIPR programs will execute efficiently. Second, many programmers are familiar with the semantics of C++, and thus will have an easier time understanding VIPR. In our representation, aspects of programs normally not visible to programmers of textual languages, such as where the return from a procedure will go, may be made explicit. While our language allows every aspect of program execution to be explicit and visual, we provide for a programming environment to hide as much detail as the user desires.

Object-oriented languages, such as Smalltalk [12] and C++ [25], facilitate programming by providing powerful features such as inheritance, polymorphism, and dynamic dispatch. Such features can greatly reduce the amount of code needed to solve a specific problem and also facilitate code reuse. However, debugging, and modifying object-oriented programs can be more difficult than doing the same things to procedural programs. Furthermore, learning how to program in an object-oriented paradigm can be very difficult. Features such as polymorphism and dynamic method lookup can reduce the readability of object-oriented programs and require the programmer to understand more complex semantics than procedural programs.

Existing solutions to some of these problems are based on sophisticated programming environments that help reduce and resolve ambiguities that arise. Some of these tools, such as browers, are based on simple visualizations of some part of the program (e.g., the structure of the class hierarchy). In this paper, we argue that a completely visual object-oriented programming language, VIPR (Visual Imperative PRogramming), has significant advantages over textual object-oriented languages.

The design of VIPR was inspired by Kahn's Pictorial Janus [15]. In VIPR, programs are represented as nested circles and language features such as continuations and dynamic dispatch, which exist implicitly in the textual representation, can be made explicit. Our language can be characterized succinctly as viewing a program flowchart from the point of view of a bead moving along the flow lines. Furthermore, although our language is two-dimensional, its visual properties suggest that it is a projection of a three-dimensional representation. In this paper, we briefly discuss the basic language features or VIPR, and then specifically describe how VIPR supports object-oriented programming with objects, classes, inheritance, polymorphism, and dynamic dispatch.

The VIPR language provides the following advantages over existing visual and textual programming languages. First, the semantics of programs written in the language may be understood in terms of a small number of graphical rewrite rules, enhancing learnability and understandability. In particular, programmers do not have to be familiar with or understand textual programming languages to write programs in VIPR. Second, because VIPR is completely visual, all aspects of the static and dynamic execution of programs in VIPR are understandable with the same visual metaphor. Thus, our framework integrates existing environment frameworks for visual debugging, browsing, and visual execution environments. Third, our language makes aspects of program execution that are normally hidden from programmers explicit. For example, the semantics of program continuations, dynamic dispatch, and class inheritance are all explicitly represented in VIPR. While making these features explicit may increase a programmer's ability to understand how they work, for an experienced programmer such information is unnecessary. Thus, we also describe how an environment for VIPR can be constructed that eliminates unwanted details in displaying the program. VIPR has significant advantages for large-scale programming because its semantics are based on topological information and as such programming environments have freedom to translate and scale parts of programs as desired to expose or hide details as necessary.

VIPR supports object-oriented programming in the following ways. VIPR defines objects and classes by aggregating definitions of data and procedures together visually. Class interfaces are defined by "hiding" all the private methods and data from clients using the classes. Only when method code defined inside a class is executed are the private parts of a class exposed. Inheritance is implemented by defining subclasses spatially inside the superclass. Polymorphism is provided by allowing the kind of object a variable denotes to change dynamically. Thus, during the visual execution of a program, the current type of a variable changes based on the object it points to. Dynamic dispatch is encoded visually, that is, a dynamically dispatched method contains branches to all the appropriate implementations of the method and the correct branch is selected dynamically based on the type of the object being called. We present example VIPR programs illustrating all of these concepts later in the paper.

We have considered several two-dimensional program representations that can be obtained by viewing a flowchart in three-dimensions and projecting it back onto two dimensions. Changing the projection used and the point-of-view of the viewer can give such a program representation interesting visual properties. We feel that the representation we have chosen has certain advantages over other choices. In particular, only a small number of simple principles are required to understand VIPR programs and we have been able to formally define them. Nevertheless, we feel that further research in this area is called for and that our research investigates only one of several possible interesting representation choices.

This paper has the following sections. Section 2 discusses why a completely visual object-oriented programming language is useful. Section 3 describes related work and Section 4 presents a overview of the basic features of VIPR. Section 5 describes specifically how VIPR supports the features required for object-oriented programming and Section 6 summarizes our thoughts.

2 Motivation

In this section, we discuss some of the problems encountered in using object-oriented languages and then describe why a completely visual object-oriented language appropriately addresses these problems.

2.1 Problems with the Object-oriented Paradigm

Object-oriented programming provides significant advantages over procedural programming, including software reuse, increased reliability, and reduction in code size. On the other hand, writing, debugging, and modifying object-oriented programs can be more difficult than doing the same things to procedural programs for a number of reasons. Furthermore, learning the semantics of object-oriented languages presents challenges to procedural programmers. Here, we outline some of the problems programmers encounter.

Object-oriented programming provides polymorphic variables. In particular, a variable denoting a superclass can at a later time be used to denote an object of any subclass. As such, when a method associated with that variable is called, the specific method will depend on the current object denoted, which will change dynamically. Thus, programmers looking at the static program text are unable to know what specific methods will be called at runtime. This situation may require care in programming because all possible runtime behaviors need to be considered to ensure that a program will execute correctly. This problem has led some researchers to wonder aloud whether using polymorphism is the modern equivalent of programming with `goto`'s [21].

Object-oriented programming requires the programmer to understand a complex execution model of method invocation. Unlike simple call/return semantics, method invocation semantics may involve dynamic method lookup, which is a concept generally unfamiliar to procedural programmers. The specific method that will be invoked when a message is sent depends on understanding both the execution model and the structure of the class hierarchy. Our experience teaching these concepts leads us to believe that many programmers have difficulty understanding them. We are aware of companies that are now expending considerable effort to retrain procedural programmers (e.g., of COBOL) to write object-oriented programs in C++.

Object-oriented programming requires the programmer to factor the problem being solved in ways that are different from procedural programming. Specifically, programmers need to determine what classes are needed, how these classes relate to existing classes, and what data and operations these classes support. While object-oriented design methods exist, there is still relatively limited experience with them as compared to more traditional methods.

Object-oriented programming often involves the creation of highly-interconnected heap data structures. Programmers encounter problems operating on (e.g., traversing, inserting into, etc.) these structures and also have difficulty correctly deallocating objects that are not longer in use.

While this problem is not unique to object-oriented programming, such programs will probably benefit more from language support for visual debugging of heap structures..

2.2 Weaknesses with Existing Solutions

Historically, sophisticated programming environments have been employed to reduce the difficulty of writing, understanding, and modifying object-oriented programs. Such environments may include class browsers to see the high-level organization of the class hierarchy, data structure visualization support, and domain-specific execution visualizations. In this way, all aspects of a program including the static text, the data structures, and the dynamic control flow can be visualized to a limited extent. We address each of these tools in turn and discuss their individual weaknesses.

Class Browsers These tools provide a static view of a program's class hierarchy (e.g., [13]). Browsers are not intended to be used to understand the dynamic behavior of a program and the browsing metaphor does not extend to understanding other aspects of the static program structure. Thus, a programmer requires additional environment support beyond browsers to understand, modify, and debug object-oriented programs.

Data Structure Visualizers These tools range widely in their complexity and include extensions to debuggers to display lists or trees (e.g., [18, 19]). As an example of a text-based data structure visualizer, almost every Lisp environment provides automatic code and data pretty-printing support. Common Lisp has such support built into its definition [24].

The problem with this approach is that one can only view program data with such tools and not the program itself (except, to a limited extent, in Lisp, where program and data have the same form). Also, since the tools are often wedded closely with the debugger, they tend to be ad hoc extensions of a mostly textual debugging paradigm.

Domain-Specific Execution Visualizations These tools include toolkits for building domain-specific visualizations of the execution of programs, including object-oriented programs. With such toolkits, programmers can visualize state-transition diagrams, oscilloscopes, logic circuits, or any of a number of simulation applications (e.g., [17]). These toolkits are ad hoc because they do not provide a general framework for understanding the behavior of programs, but instead allow applications in particular domains to be visualized.

A weakness of all of these approaches is that they usually stand alone and are not designed to be coordinated with each other. What is missing is a standard visual framework for understanding the program, its data, and its dynamic execution through a single integrated tool. We seek to unify static and dynamic aspects of a program and its data with a visual metaphor, just as the form of program and data are unified in Lisp with a textual metaphor.

While environment support is very useful and necessary, we feel that a more appropriate approach to increasing the comprehensibility of object-oriented languages is to program and execute the languages using a completely visual metaphor. This approach allows all the tools described above to work together in a unified framework, which has distinct advantages. First, the programmer avoids having to translate back and forth between the different representations used to present the static and dynamic forms. Second, the programmer avoids the need to learn a number of different ad hoc program representations and instead learns only one. Finally, the semantics of a completely visual language are described entirely by a small number of graphical rewrite rules, and as such are relatively easy to learn.

3 Related Work

In this section we describe other work in completely visual programming languages and also other work in visual object-oriented programming languages.

3.1 Completely Visual Languages

Although much visual language work has concentrated on visual models of basically textual languages, in which the semantics of the visual language derive from the semantics of the underlying textual language (and not from the graphical properties of the visual representation), a few visual languages have been designed whose semantics derive entirely or predominantly from graphical rules. The most significant such language is Pictorial Janus [15], which was originally designed to model the execution of the constraint logic programming language Janus, but whose execution semantics may be derived from graphical rules applied to the visual representation. Kahn described such languages as "completely visual."

Pictorial Janus is unique in that a snapshot of the dynamically executing program contains a complete copy of the static program being executed. Kahn's definition of completely visual languages included this requirement, but if we relax the definition to allow languages where the state and the program are separate, another class of languages may be considered completely visual: the graphical transformation languages. In these languages, a program consists of a set of before/after pairs of diagrams. State consists of a set of graphical entities and their relationships. If the "before" part of a before/after pair matches part of the state, the state is transformed to conform to the "after" part of the pair. BITPICT [11] is one of the simplest of such languages, in that its before/after pairs are simple pixel patterns. ChemTrains [1] and Vampire [16] allow more complex visual entities and relations, and also permit variables in the transformation rules.

Completely visual languages of both types have the advantage that the user does not need to understand two different (albeit related) sets of semantics, the semantics of the visual model and that of the underlying textual model, but can understand and write programs with knowledge of only the visual semantics.

3.2 Visual Object-Oriented Languages

A number of visual object-oriented environments have been proposed, although none of them have been completely visual. In fact, most of them simply visualize objects, but set aside the problem of how methods should be visualized.

Rogers [23] proposed an object-oriented visual language in which classes and methods are defined in terms of relations between objects. This language does employ visual representations of methods, but the relational approach taken in this system does not appear to be applicable to the visual modeling of an imperative language.

More commonly, visual object-oriented environments use visual representations to define the relationships between classes, and to create new classes by assembling prototypes of previously defined classes. ObjectWorld and Rehearsal World are typical systems of this type. Penz's ObjectWorld [20] does model an imperative object-oriented language, however, only the objects themselves are represented visually. The methods by which they interact with each other are defined in Smalltalk. These methods are displayed in pop-up windows, which may be hidden while the program executes. Finzer and Gould's Rehearsal World [10] is a similar Smalltalk-based visual environment in which the objects, but not the methods, are visualized. In Rehearsal World, however, methods may be defined by demonstration as well as textually, but the representations of methods maintained by the system are textual Smalltalk.

Demonstrational systems are a natural application for visual object-orientation. Borning's ThingLab [2] is probably the most famous such system. ThingLab employs a constraint model rather than an imperative model, however, and while the constraints may be represented graphically [3], the methods for resolving the constraints are displayed textually.

A number of other visual languages provide object-oriented features. Prograph [8, 9], for example, provides a class system where the hierarchy is presented graphically. Inheritance, instance variables, dynamic dispatch, and polymorphism are all provided in Prograph, although dynamic dispatch and polymorphism are not shown graphically. Forms/3 [4, 5] , although not explicitly an object-oriented language, offers data abstraction and polymorphism. Class inheritance is not provided.

4 An Introduction to VIPR

VIPR is a programming language and not a program representation. This distinction is important to understand, as graphical representations of textual languages are sometimes supported in CASE tools. The distinction between existing graphical representations and VIPR is that VIPR is intended to be an executable specification. That is, programs written in VIPR are intended to be executed and we have defined graphical rewrite rules that define the semantics of that execution [7]. Unlike VIPR, program representations are not executable and are simply intended to make some aspect of a textual language more understandable.

One reason that VIPR may be confused with program representation methods is because we intentionally have defined VIPR to have semantics similar to familiar textual languages, such as C++. As such, we will describe the semantics of VIPR constructs by defining them in terms of their C++ equivalent. This use of C++ is solely for purposes of readability; as we have mentioned, one of the most significant features of VIPR is that its semantics may be defined exclusively in terms of graphical transformation rules, with no reference at all to any other textual language.

4.1 Control Constructs

Although the control constructs of VIPR are interesting in themselves (see [6] for details) a detailed discussion is outside the scope of this paper and we only provide sufficient information to appreciate the unusual aspects of a completely visual programming language and to understand the subsequent discussion of objects and how they are handled in the language.

The design of VIPR is closely related to Kahn's Pictorial Janus [15]. While Pictorial Janus visually models constraint logic programming, and therefore makes allowances for nondeterminism, VIPR is intended to model conventional imperative constructs (with the addition of objects and class inheritance). It is modeled after C++, and is statically scoped and typed.

To force sequentiality and determinism in our model, we have modified the Pictorial Janus model to require the participation of a "state" object in every graphical transformation (each of which corresponds to a computation step), and we then provide a single state object to participate in the first step.

A program in VIPR is represented as a set of nested rings (see figure 1). A computation step is represented by the combination (merging) of two such nested rings, which may only combine when a state object is present and attached to the outermost of the merging rings. A ring contains an optional textual condition, which may reference state, and which must be true for the ring to participate in a merging. (If no condition is present, the default condition is true.) If there are two or more rings nested inside a ring with an attached state object, the conditions of all these rings are evaluated, and one of the true rings will be evaluated. (If more than one ring has a true condition, the result is undefined.) Figure 2 illustrates what an if-statement looks like in VIPR.

The result of merging two rings is to execute the action, generally defined textually, associated with the innermost of the two rings being merged. These actions are usually assignments or input/output statements. We can therefore view execution as a series of rings merging outward from the center and progressively altering program state by combining with the outermost ring. Alternatively, we can visualize the nested rings as a pipe receding into the distance, and execution as traveling down the pipe, choosing branches based on conditions written on the inside of the pipe at branch points, and carrying the state with us so that we can modify it based on actions written on the inside of the pipe as we pass.

Some simple examples provided in figures 1–3 illustrate the flavor of control constructs in VIPR. For each construct, we present the equivalent textual language construct, although the semantics of VIPR constructs can be understood without any reference to, or even knowledge of, textual languages. The boldface text present at the bottom of the figures is not present in VIPR programs—we present it only to clarify what construct is being illustrated. Figure 1 illustrates sequential constructs, figure 2 illustrates conditionals, and figure 3 illustrates iteration. In figures 2 and 3, note the arrows connecting one ring to another. These arrows represent substitution and indicate that the construct at the destination end of the arrow may replace the ring at the source end.

4.2 Procedures and Functions

Procedures and functions are sets of nested rings outside the main procedure's rings (figure 4), and therefore outside of the normal linear course of execution. A procedure call is represented by a ring possessing an arrow pointing to the procedure being called. Since the arrow represents substitution, a copy of the procedure may then be substituted for the procedure. Nodes attached to the procedure call represent arguments, and corresponding nodes attached to the outermost ring of the procedure represent parameters. Each procedure must have at least one parameter, representing the continuation, or return address. A procedure call must have a corresponding argument containing an arrow pointing to the statement to be executed after the procedure call returns. A procedure

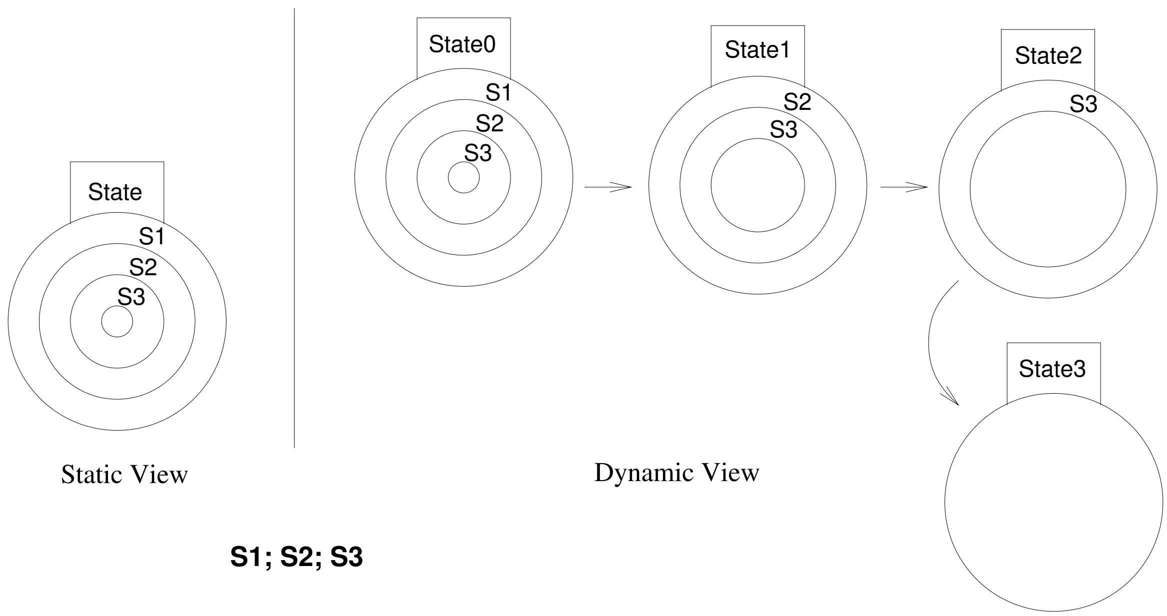

Figure 1: Static and Dynamic VIPR Representations of Sequential Statements S1, S2, and S3

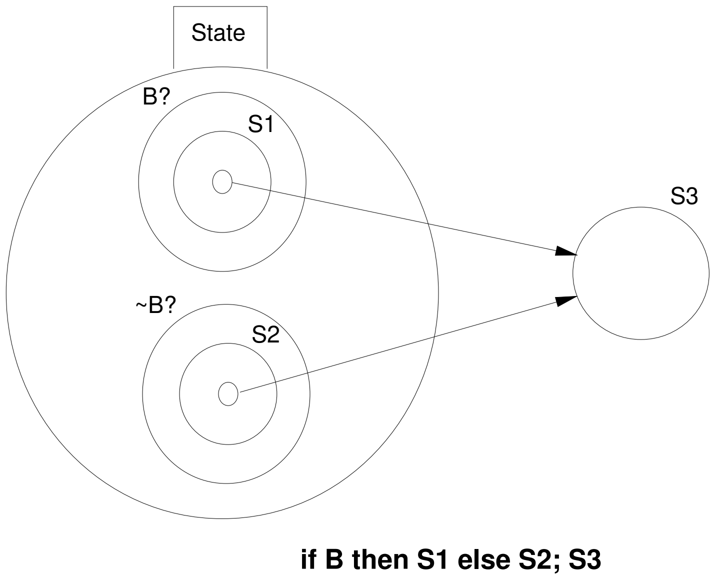

Figure 2: Static VIPR Representation of an If Statement

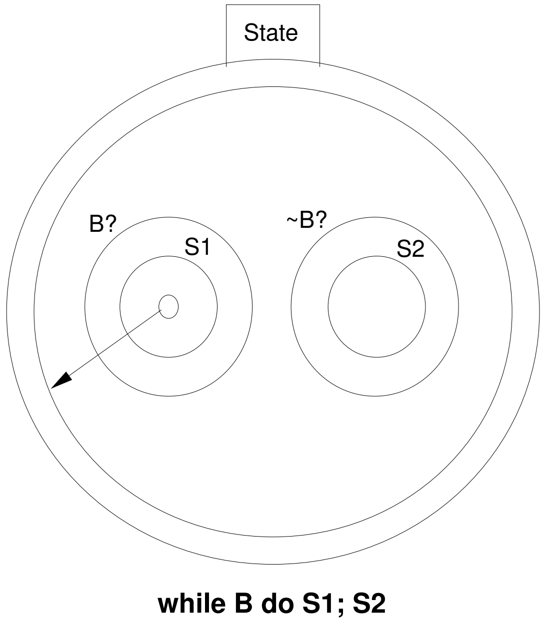

Figure 3: Static VIPR Representation of a While Statement

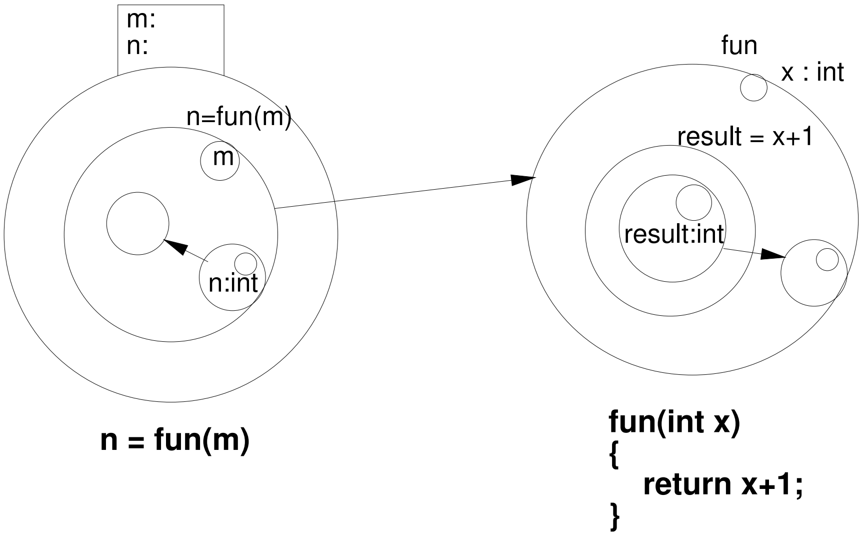

Figure 4: Static VIPR Representation of a Function Definition and Call

return statement is a ring with an arrow pointing to the continuation parameter. It indicates that return may be performed by substituting the continuation value for the return statement ring.

Procedure calls can be viewed as graphical substitution. An arrow represents the substitution of a copy of the thing pointed to for the thing from which the arrow originates. Alternatively, procedures may be visualized as pipe segments which are spliced into the mainline pipe in place of the procedure call segments.

Parameter passing and function return values are also handled graphically. A return continuation is simply a specialized form of parameter, and generalized parameters may also be represented as small circles within the rings representing procedures and procedure calls. Parameters are matched either positionally, with respect to their position in terms of the return continuation which is always present, or by name, where optional parameter names can be provided. At call time, when a copy of the procedure body is substituted for the call, the argument rings and the parameter rings merge and whatever values corresponded to the arguments now correspond to the parameters. All arguments are passed by value, although pointers (represented by arrows) may be explicitly passed to simulate call by reference.

Return values are passed back through the return continuation. The statement following the procedure call (that is, inside the call ring) possesses a small ring representing the return value. Correspondingly, there are small rings associated with the function's return continuation parameter and the ring representing the return statement. All of these larger rings are linked by arrow chains representing substitution (that is, the return continuation parameter gets substituted for the return statement, and the statement to be executed after the call gets substituted for the return continuation parameter), and in the course of these substitutions the return value gets passed back to the caller as the corresponding small rings are merged. Note also in the example that the argument (`x`) and return value (`result`) are both typed as integers. Such type annotations are included to provide more static information about the behavior of functions and procedures. The fact that VIPR is statically typed is a preference of the designers—we feel that static type indications will generally lead to more robust programs.

5 Object-oriented Programming in VIPR

In this section, we describe how VIPR supports all the mechanisms commonly found in object-oriented languages. We illustrate these ideas with compete, simple examples (i.e., point and shape classes), that are commonly used to describe object-oriented features. At the end of the section, we discuss what we expect a programming environment for VIPR to include and indicate why we believe that VIPR will scale to solving larger, more complex problems.

5.1 Objects and Classes

Many object-oriented languages provide classes, which are a means of describing a collection of objects with common characteristics, and objects, which represent instances of the classes. Objects usually contain data and operations that manipulate that data (instance variables and methods are the terms commonly used). Furthermore, classes also provide a means of encapsulation and information hiding (e.g., private instance variables and methods in C++ classes). VIPR describes classes by aggregating data and methods together physically and enclosing them. In this section, by example, we illustrate how VIPR supports all of these features.

First consider the C++ definition of a very simple point class presented in Figure 5. This example illustrates many of the issues related to objects and classes as outlined above. Figure 6 shows the same class defined in VIPR. As the figure shows, the VIPR definition groups methods and instance variables by enclosing them with a dotted line. In essence, a VIPR class is a grouping of individual methods, each of which looks like a strand of fiber being viewed head-on. Thus, VIPR's classes look like fiber bundles with some data associated with them. Note that in VIPR, as opposed to C++, all objects are heap-allocated and thus all variables denote references to objects.

VIPR also supports information hiding and the separation of the class interface from its implementation. In VIPR, instance variables or methods that are private to a class are indicated by enclosing them with double boxes or circles (e.g., the variables `x` and `y` in the point class are private).

To see how these private values and implementation details are hidden from the users of a class, we must see how objects and classes in VIPR are declared, created, and used. Consider the C++ code fragment in figure 7, which allocates two point objects, moves them, and computes the distance between their "`x`" coordinates. The following figures show the visual execution of the VIPR version of this program. In particular, we show the state of the program at Statements 1, 2, 3 and 4 as indicated by the C++ comments in the text. Note that Statement 3 is inside the `xDistance` method in figure 5.

Figure 8 shows the VIPR program before execution has begun (i.e., at statement 1). The variables `p1` and `p2`, declared to be of type point, have not been given values yet. As such, they currently denote pointers to small instances of the VIPR point class that are shaded, indicating that the instances are not actual objects (we call such instances *pseudo-instances*). Also note that only the public parts of the point class definition are visible to users of the class. Only when methods

```
class point {
    int x, y;
 public:
    void moveTo(int newx, int newy)
      { x = newx; y = newy; }

    int xDistance(point* p)
      {
        // Statement 3
        return p->x - x;
      }
};
```

Figure 5: C++ Point Class

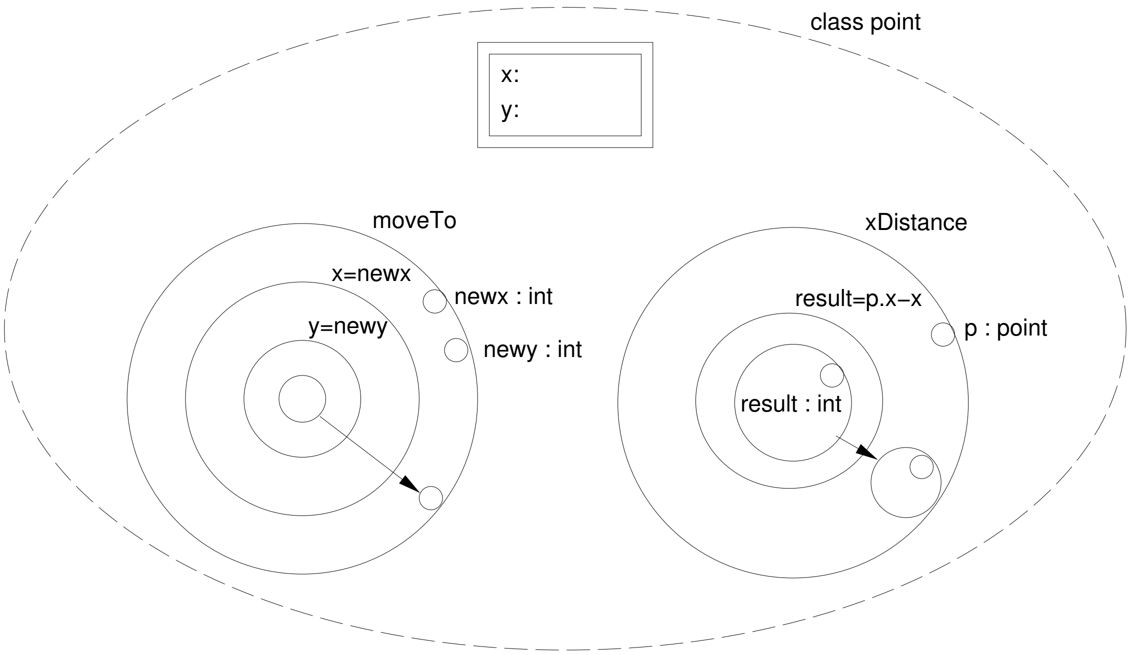

Figure 6: VIPR Point Class

```
main()
{
    point *p1, *p2;
    int d;
                                // Statement 1
    p1 = new point;
    p2 = new point;
    p1->moveTo(3,4);
    p2->moveTo(0,0);
                                // Statement 2
    d = p1->xDistance(p2);
                                // Statement 4
}
```

Figure 7: C++ Program Using the Point Class

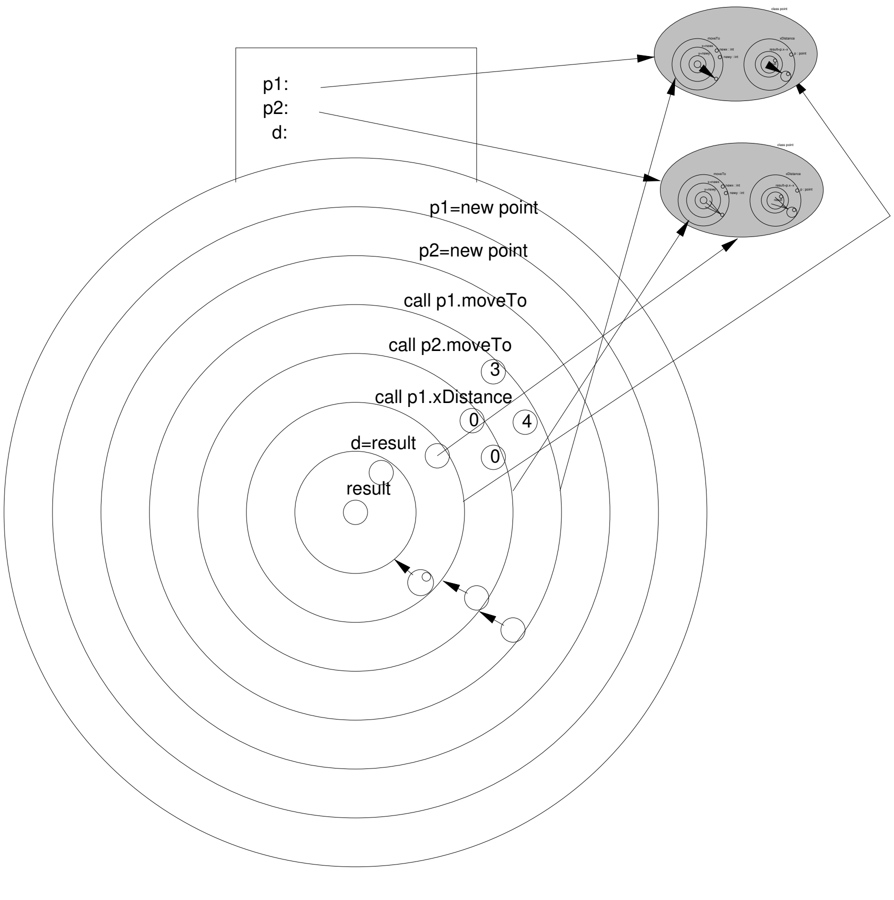

Figure 8: VIPR Program Using the Point Class (at Statement 1)

of the class are entered are the private portions of the class definition made visible. The figure also indicates what the statements in the main program look like in VIPR. Because the program is entirely sequential, the main program is denoted by a set of nested rings, as mentioned. Each ring has an associated action, such as assignment (e.g., `p1 = new point`) or a method call (e.g., `p1.moveTo(3, 4)`). In the figure, we see the arguments to the methods being passed by matching argument indicators, and we also see a return value being returned from the xDistance method. Later figures illustrate how this program looks when it is being executed.

Figures 9, 10, and 11 show the VIPR program just before the call to `p1.xDistance`, just inside the `xDistance` method, and just after the return from the `xDistance` method, respectively.

Figure 9 shows the VIPR program after the variables `p1` and `p2` have been given values (i.e., at statement 2). Now the pseudo-instances of the point class have been replaced by actual instances. The two calls to `moveTo` have also been executed, and so the instances have been given specific values for "`x`" and "`y`" (although the values are not currently visible). In figure 9, we now see the call to `xDistance` being set up, with object `p2` as the parameter to the method. The call also clearly shows that the method returns a value (indicated by the parameter being passed back to the continuation parameter).

In figure 10, we have now entered `p1`'s `xDistance` method, which has been expanded to provide more detail. Because we have entered `xDistance`, the scope has changed. Thus, the private instance variables `x` and `y` are now part of the environment, as well as `self`, the name of the object whose method is currently executing, as well as all the parameters to the current method (in this case, the point `p`). Furthermore, the names `p1`, `p2`, and `d`, local to the main procedure, are no longer available. We also see that the text `p.x` in the method now has specific meaning because the parameter `p` is now bound to a specific point object. Note that `result` is the name given to the return value of the method, indicated by the small circle attached with the innermost ring, that is matched with the continuation parameter when the method returns.

In figure 11, we see the program after `p1.xDistance` has returned. The return continuation of `xDistance` provides a place and a name for the return value (in this case -3) and the final operation the program performs is to assign that value to the variable `d`. Note that because we have returned to main, the scope has changed back to the original scope.

This example has served to illustrate how VIPR supports objects, classes, and all the semantics related to creating, defining, and manipulating objects.

5.2 Inheritance, Polymorphism, and Dynamic Dispatch

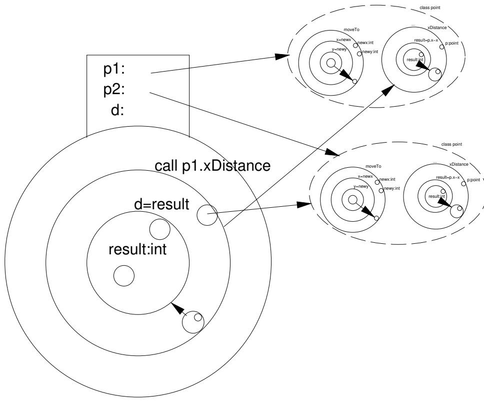

Figure 9: VIPR Program Using the Point Class (at Statement 2)

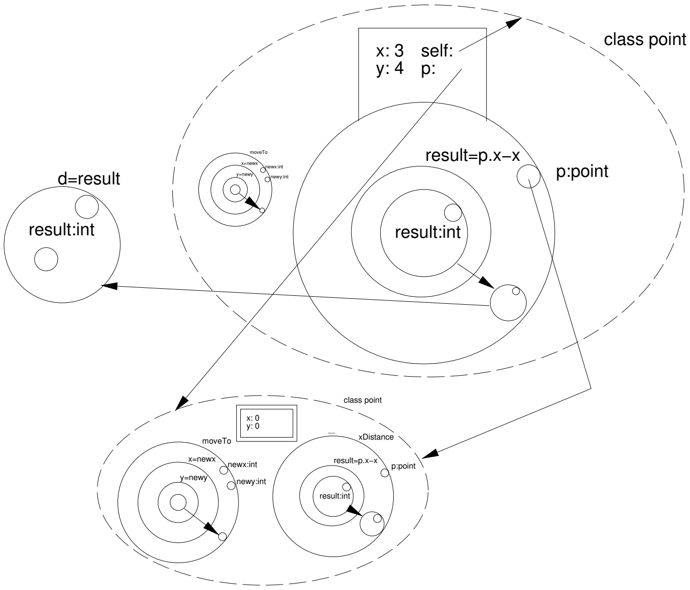

Figure 10: VIPR Program Using the Point Class (inside xDistance)

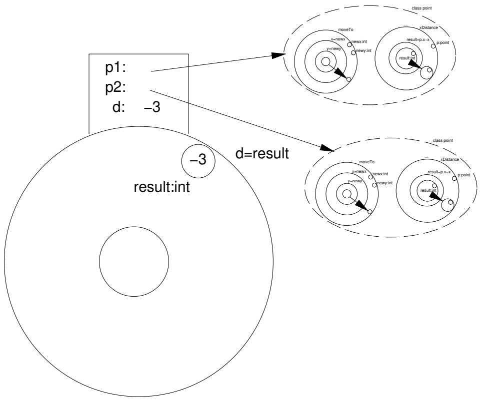

Figure 11: VIPR Program Using the Point Class (at Statement 4)

```
class shape : public point {
public:
    virtual double area() = 0;
};

class square : public shape {
    int side;
public:
    double area() { return side * side ; }
};

class circle : public shape {
    double rad;
public:
    double area() { return 3.14 * rad * rad ; }
};
```

Figure 12: C++ Shape, Square, and Circle Classes

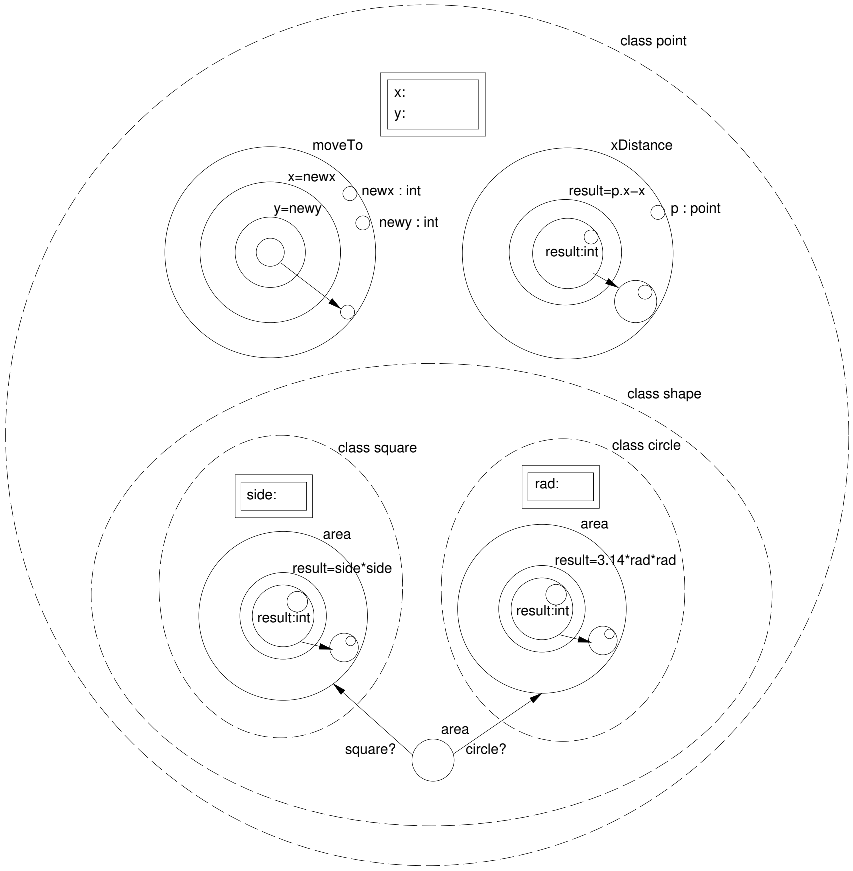

Figure 13: VIPR Point, Shape, Square, and Circle Classes

To show how VIPR supports inheritance, polymorphism, and dynamic dispatch, we will extend the example provided in the previous section to include these features. Specifically, we consider subclasses of the point class that implement objects of different shapes. The C++ definition of these subclasses is shown in figure 12. Note that the method `point::area` is defined to be "= 0" indicating that subclasses of point are required to define this method appropriately. The VIPR definition of these classes is shown in figure 13. Shape is a class with two subclasses, square and circle, each of which have private instance variables. VIPR represents this relationship by including the definition of the shape class inside the point class. Spatial inclusion is used to represent subclass relations because it allows the semantics of inheritance to be spatially suggested. Specifically, the implication that superclasses "include" subclasses spatially suggests the subtype principle.

Using spatial inclusion has some costs. In particular, adding new subclasses requires modifying the superclass (by defining the subclass inside it). Fortunately, VIPR's semantics allow automatic translation and scaling, and so the user would not be responsible for reorganizing the contents of a class when defining a new subclass. We also note that VIPR supports only single inheritance. Thus the spatial inclusion metaphor is appropriate for VIPR. If we had chosen multiple inheritance semantics, using inclusion would be problematic. We leave a visual representation of multiple inheritance semantics in VIPR for future work.

The `area` method is a dynamically dispatched method defined in shape that is redefined both in square and circle. Calls to `area` may be calls to either `square::area` or `circle::area`, depending on the type of the object invoking the method. This relationship is indicated by showing the area method with type conditions indicating that either the square or circle method may actually be invoked at runtime. Thus, the behavior of dynamically dispatched methods is explicit and very clear from the visual representation. Note that this representation does not denote or imply a specific implementation (i.e., using vtables as C++ does). Instead, it represents the underlying semantics of dynamic dispatch explicitly.

To understand how a VIPR program uses polymorphism and dynamic dispatch, we include an example program that sums the areas of objects in a polymorphic array of shapes. Figure 14 shows the C++ version of this program, while figure 15 shows the static representation of the VIPR equivalent. The program initializes a loop index, `i`, and the elements of the polymorphic `shapes` array. It then loops over the elements of the array, summing the areas of all the shapes in it. The VIPR code illustrates the use of a conditional to test the loop index and the looping embodied by the arrow jumping out of the loop body. Because the array "`shapes`" and variable "`s`" have not been initialized, we see that they currently indicate a pseudo-instance of the shape class, with all possible subclasses available.

```
main() {
    int i = 0;
    double sum;
    shape *shapes[2], *s;

    shapes[0] = new square;
    shapes[1] = new circle;

    while (i < 2) {
        s = shapes[i];
                                  // Statement 1
        sum += s->area();
        i += 1;
    }
    cout << "Total Area = " << sum << "\n";
}
```

Figure 14: C++ Program using Polymorphic Objects with Dynamic Dispatch

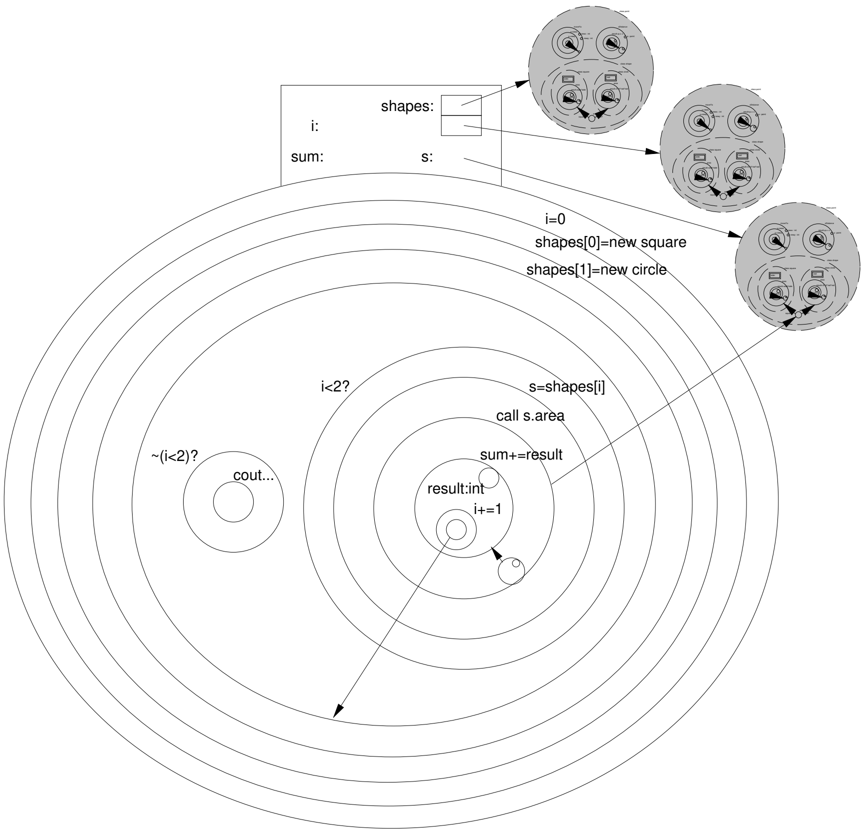

Figure 15: VIPR Program using Polymorphic Objects with Dynamic Dispatch

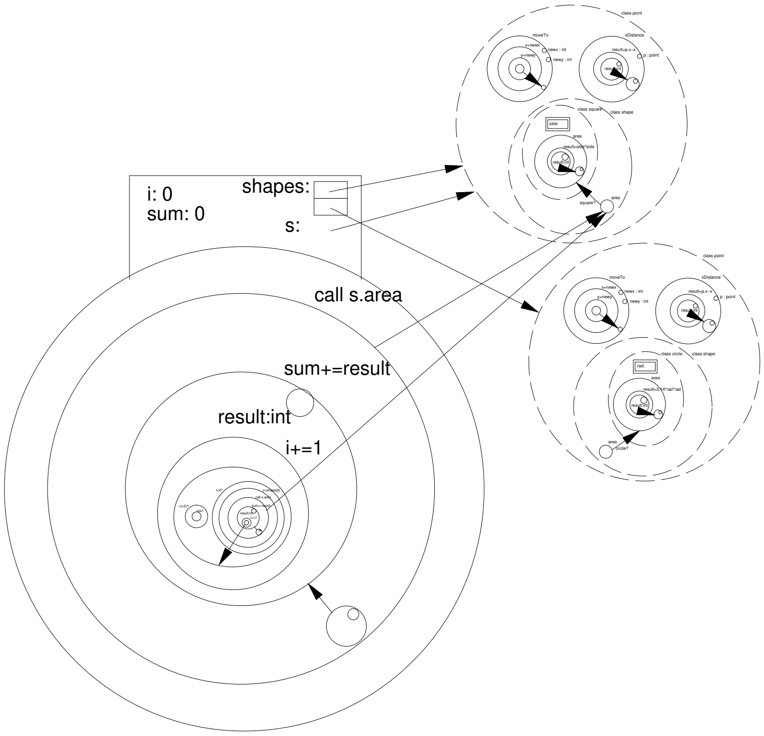

Figure 16: VIPR Program Using the Shape Class (Statement 1 with i = 0)

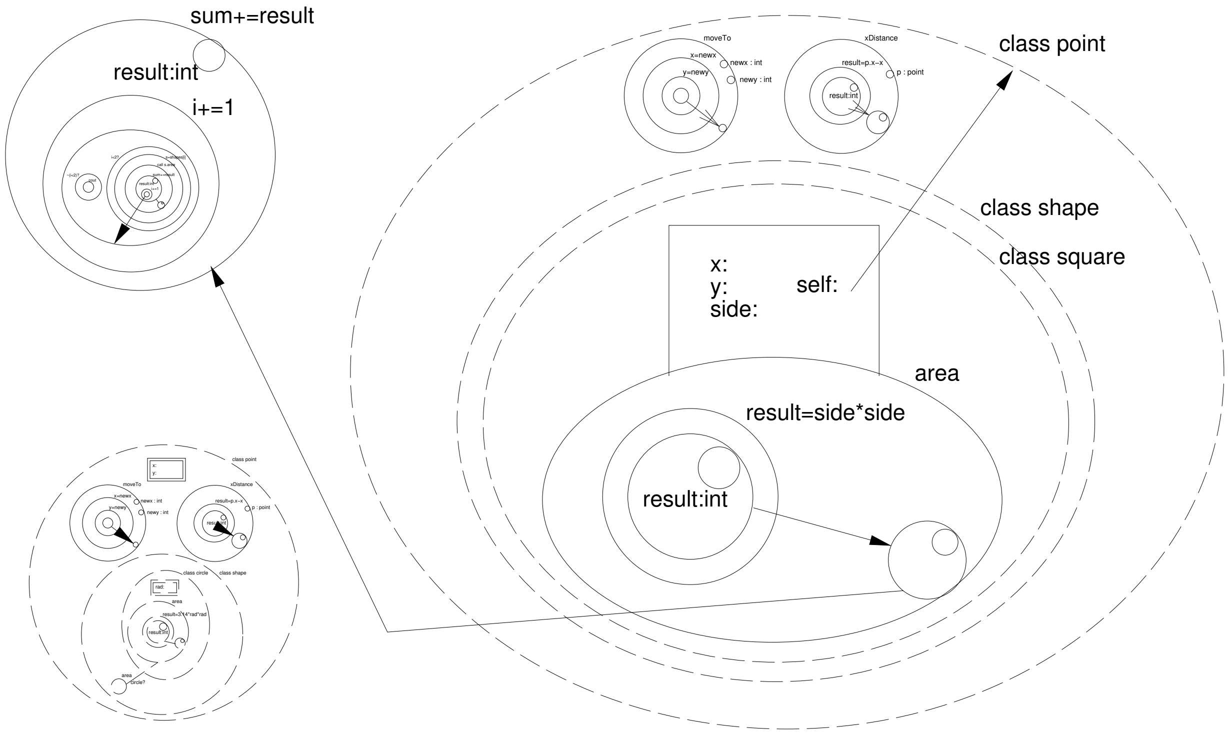

Figure 17: VIPR Program Using the Shape Class (inside Square area method)

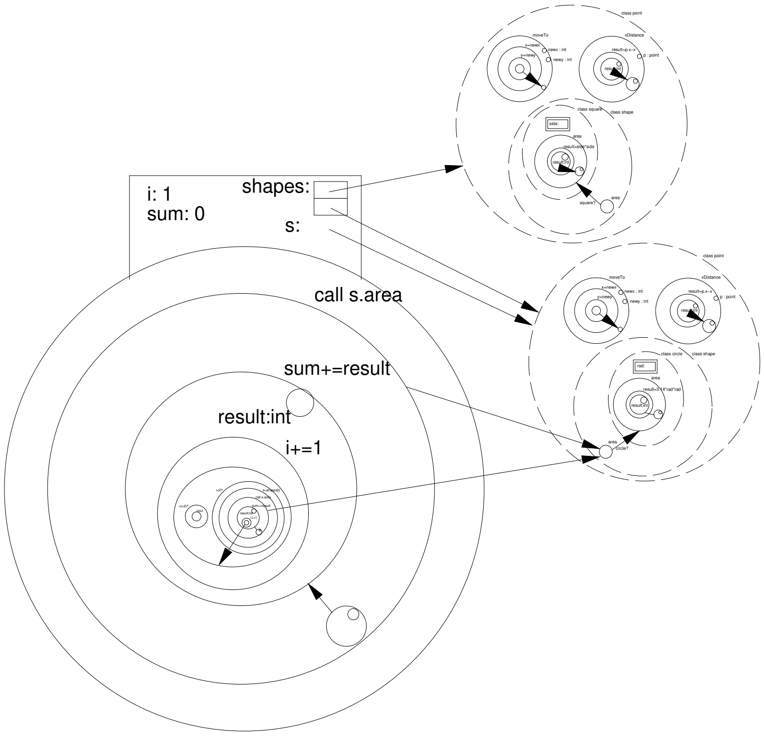

Figure 18: VIPR Program Using the Shape Class (Statement 1 with i = 1)

Figures 16, 17, and 18 show the dynamic execution of this program at Statement 1 (indicated in the C++ text), inside the area method of the square instance, and at Statement 1 again (with i = 1), respectively. These snapshots illustrate how polymorphic variables can point to objects of different types and how dynamically dispatched methods in these objects are actually dispatched.

Note that in figure 16, the **shapes** array has been initialized and each element points to a different shape subclass. Because the specific subclass of each array element is now known, we see that the thumbnail diagram of each instance has been refined to show only the methods and instance variables associated with the particular subclass. Also, we see that dynamically the area method for each array element has been resolved and so no dynamic dispatch is no longer necessary. Again, while the underlying implementation may not behave in this way (i.e., resolve the dynamic dispatch by eliminating the indirection), the semantics of assigning a specific object type to the array elements are explicitly shown. Figure 16 also shows the variable "**s**" pointing to the first object in the array. Thus, when the **s->area** method is called in the next statement, the **square::area** method will be invoked. Finally, figure 16 shows that the body of the while-loop has been replicated in the innermost ring of the loop, indicating another unfolding of the loop has occurred.

In figure 17, we have just entered the **square::area** method. As with the **xDistance** method in the previous example, this figure illustrates the change of environment that occurs when a method is invoked. Similarly, the `square::area` method returns a value via a return continuation.

In figure 18, we are again at Statement 1, but now "`i`" has the value 1. In this case, the variable "`s`" denotes a circle object and the `circle::area` method is about to be invoked. As in the previous diagram, the next iteration of the loop has been unfolded by applying the arrow substitution rule (i.e., the destination of an arrow can be used to replace the contents of the source ring of the arrow).

5.3 Scalability and Environment Support for VIPR

The examples we have used to illustrate the object-oriented features of VIPR are quite small; the visual complexity of these examples must immediately raise some concern about the scalability of programs written in VIPR. To address concerns of scalability, we feel that VIPR will naturally be used in a programming environment specifically designed to support the creation of large VIPR programs. Furthermore, we feel that properties of VIPR's representation allow VIPR programs to be easily manipulated by such an environment. Specifically, because VIPR's semantics are based entirely on topological relationships (e.g., containment, connectedness), VIPR programs can be dramatically transformed without modifying their meaning. Specifically, parts of the program can be greatly expanded while other parts are shrunk down. Procedures, functions, and objects can be translated and scaled arbitrarily as long as the connections between them are maintained. The previous examples in this section illustrate such scaling to some extent—depending on what part of the program is currently executing, that part should be enlarged to make it easier to see. Likewise, instances of objects that are not currently being used remain at thumbnail sketch size until they are needed.

We also anticipate that the environment will smoothly animate transitions between computation steps (i.e., growing and shrinking elements as necessary) in order to make the visualization of execution easier to follow. Investigators have noted the importance of smooth transitions and animation in dynamic visualizations [22] and Kahn incorporated it into his Pictorial Janus system [14].

Another problematic aspect of the VIPR language is the use of arrows. Arrows represent connections between things and too many arrows make a program confusing. We anticipate that a support environment will allow connections indicated by arrows to also be indicated using textual names. We do not dislike the use of text in programs and we believe that text should be included in VIPR programs any place where it simplifies the representation. Note that the use of textual variable names throughout the examples is an instance of such a use of text. Each variable name in fact represents an arrow to the storage associated with that variable. While the VIPR program has meaning without the variable names, it is clearly more readable, at least to an experienced programmer, if variable names are used. Likewise, the semantics of expressions can be specified completely visually in VIPR (because expressions are just a composition of functions), but for the most part expressions such as "`a + b`" are easier to understand in their textual form.

We also expect that a VIPR environment will leave out certain connections most of the time, unless the programmer explicitly requests them. For example, if there were 10 "`area`" methods in our example, the lines connecting the dynamically dispatched method to each of the implementation methods could be elided. The connections between function and procedure calls and the function being called could also be elided in the same way.

Another kind of support the environment can provide is related to viewing class hierarchies. For example, suppose a programmer only wants to see the instance variables and methods related to the `square` class in figure 15. In this case, all instance variables and methods related to "non-square" classes could be eliminated from view. In general, we expect the environment to support a number of different views of the program and that these views would be based on the visual framework provided by VIPR.

Finally, environmental support in VIPR will provide techniques such as graying-out pseudo-instances to provide clues about the status and accessibility of objects and object methods. We will count on the environment to support the separation of program interfaces from implementations by physically hiding information that is unavailable, as we have indicated in the examples. Likewise, while VIPR makes explicit many aspects of programs that are implicit in textual languages (i.e., continuations, environments, etc), we will allow the environment to hide this information from programmers if they choose not to see it. Thus, expert programmers who will not benefit from the explicit representation of this information will not suffer from its presence either.

We believe that all these techniques will improve the scalability of VIPR to practical programming tasks.

6 Summary

Object-oriented programming languages provide powerful features such as objects, classes, inheritance, polymorphism, and dynamic dispatch to make solving large, complex programming problems easier. These same features, however, make the languages more difficult to learn, and make programs more difficult to understand, modify, and debug. To address these problems, we have described VIPR, a completely visual object-oriented programming language. Succinctly, the VIPR visual metaphor is that of viewing a program flow-chart from the point-of-view of a bead flowing along the control lines.

VIPR offers the following advantages over textual object-oriented languages. First, the semantics of VIPR programs can be completely understood in terms of easy-to-understand graphical rewrite rules such as merging of adjacent rings and substitution by following arrows. Second, VIPR provides a unifying framework for visualizing static and dynamic program execution upon which a support environment can be designed based on the metaphor provided. Because VIPR semantics are topological, transformations of programs involving scaling and translation will not change the programs' semantics. Third, our language can, but does not have to, make aspects of program execution that are normally hidden from the programmer visible. The explicit representation of concepts such as dynamic dispatch, continuation passing, and environments, will allow programmers unfamiliar with these concepts to more easily understand program semantics. Finally, we have discussed how a language environment for VIPR can reduce the visual complexity of VIPR programs and hide information that the programmer does not desire to see.

7 Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments. We would also like to thank Clayton Lewis for his helpful suggestions.

References

[1] B. Bell and C. Lewis. ChemTrains: A language for creating behaving pictures. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 188–195, Bergen, Norway, 1993.

[2] A. Borning. Programming language aspects of ThingLab. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.

[3] A. Borning. Defining constraints graphically. In *Proceedings of CHI'86—Human Factors in Computing*, pages 137–143. ACM Press, 1986.

[4] M. Burnett. *Abstraction in the Demand-Driven, Temporal-Assignment, Visual Language Model.* PhD thesis, University of Kansas, 1991.

[5] M. Burnett. Types and type inference in a visual programming language. In *Proc. IEEE Symposium on Visual Languages*, Bergen, NORWAY, 1993. IEEE Computer Society Press.

[6] Wayne Citrin, Michael Doherty, and Benjamin Zorn. Control constructs in a completely visual imperative programming language. Technical Report CU-CS-672-93, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, September 1993.

[7] Wayne Citrin, Michael Doherty, and Benjamin Zorn. Formal semantics of control constructs in a completely visual imperative language. Technical Report CU-CS-673-93, Department of Computer Science, University of Colorado, Boulder, Boulder, CO, September 1993. In preparation.

[8] P. T. Cox, F. R. Giles, and T. Pietrzykowski. Prograph: a step towards liberating programming from textual conditioning. In *Proc. 1989 IEEE Workshop on Visual Languages*, Rome, ITALY, 1989. IEEE Computer Society Press.

[9] P. T. Cox and T. Pietrzykowski. Using a pictorial representation to combine dataflow and object-orientation in a language independent programming paradigm. In *Proc. Intl. Computer Science Conference*, Hong Kong, 1988.

[10] W. Finzer and L. Gould. Programming by rehersal. *Byte*, pages 187–210, June 1984.

[11] G. W. Furnas. New graphical reasoning models for understanding graphical interfaces. In *Proceedings of CHI'91*, pages 71–78, Anaheim, CA, April 1981. ACM Press.

[12] Adele Goldberg and David Robson. *Smalltalk-80: the Language and Its Implementation.* Series in Computer Science. Addison-Wesley, Palo Alto, CA, 1983.

[13] V Haarslev and R. Moeller. A framework for visualizing object-oriented systems. In *Proceedings of ECOOP/OOPSLA'90*, pages 237–244, October 1990.

[14] K. M. Kahn. Towards visual concurrent constraint programming. Technical Report SSL-91-092, Xerox PARC, Palo Alto, CA, 1992.

[15] K. M. Kahn and V. A. Saraswat. Complete visualizations of concurrent programs and their executions. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 7–15, Skokie, IL, October 1990.

[16] D. W. McIntyre and E. P. Glinert. Visual tools for creating iconic programming environments. In *Proceedings of the 1992 IEEE Workshop on Visual Languages*, pages 162–168, Seattle, WA, September 1992. IEEE Computer Society.

[17] Ross P. Morley, Pieter S. van der Meulen, and Peter Baltus. Getting a GRASP on interactive simulation. In Raimund K. Ege, editor, *Proceedings of the SCS Multiconference on Object-Oriented Simulation*, pages 151–157, Anaheim, CA, January 1991.

[18] B. A. Myers. Incense: A system for displaying data structures. In *Computer Graphics: SIGGRAPH '83 Conference Proceedings*, pages 115–125, July 1983.

[19] B. A. Myers, R. Chandhok, and A. Sareen. Automatic data visualization for novice pascal programmers. In *Proceedings of the 1988 IEEE Workshop on Visual Languages*, pages 192–198, Pittsburgh, PA, October 1988. IEEE Computer Society.

[20] F. Penz. Visual programming in the ObjectWorld. *Journal of Visual Languages and Computing*, 2:17–41, March 1991.

[21] Carl Ponder and Bill Bush. Polymorphism considered harmful. *SIGPLAN Notices*, 27(6):76–79, June 1992.

[22] George G. Robertson, Jock D. Mackinlay, and Stuart K. Card. Cone trees: Animated 3D visualizations of hierarchical information. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI'91)*, pages 189–194, 1991.

[23] G. Rogers. Visual programming with objects and relations. In *Proceedings of the 1988 IEEE Workshop on Visual Languages*, pages 29–36, Pittsburgh, PA, October 1988. IEEE Computer Society.

[24] Guy L. Steele, Jr. *Common Lisp: The Language.* Digital Press, 2nd edition, 1990.

[25] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Series in Computer Science. Addison-Wesley, Reading, MA, 2nd edition, 1991.