Machine Tongues 11: Object-Oriented Software Design

Stephen Travis Pope[1]

Introduction

Object-oriented programming is a term that represents a collection of new techniques for problem-solving and software engineering. Two previous articles in this "Machine Tongues" series have introduced object-oriented programming, presenting tutorials to this technology, and describing its application to music modeling and software development (Krasner 1980, Lieberman 1982). This paper discusses the new problem-solving techniques that constitute the object-oriented design methodology. Object-oriented analysis, synthesis, design and implementation are presented, while stressing the issues of design by analytical modeling, design for reuse, and the development of software packages in terms of frameworks, toolkits and customizable applications. Numerous object-oriented software description examples and architectural structures are presented including music modeling, representation and interactive applications.

The term "object-oriented programming" is currently a buzzword in much the same way that "artificial intelligence" was 5 years ago, or "structured programming" was 15 years ago. The parallel runs deeper in light of the fact that each of these three concepts represent revolutions in software engineering that encompass new programming languages, new types of tools, new models of what applications and interaction are, and new software description and design methodologies. In each case, there were also long periods of debate between so-called "purists" and those they deemed "impure" in the interpretation of their respective definitions and dogma. A lively debate is currently taking place in the object-oriented programming community as to definitions of object-oriented programming and object-oriented languages as well as metrics of object-oriented programming and object-oriented software design. Research papers from the leaders in the field with titles such as "*What is 'Object-Oriented Programming'*" (Stroustrup 87), "*What Object-Oriented Programming May be—and What it does Not Have to Be*" (Madsen and Moeller-Pedersen 1988); and "*Dimensions of Object-based Languages*" (Wegner 1986), have surfaced in the recent literature. Another title that underscores the depth of the difference between object-oriented programming and "traditional" programming is "*Teaching Object-Oriented Programming is More than Teaching Object-Oriented Programming Languages*" (Knudsen and Madsen 1988).

This essay will outline object-oriented problem-solving and software design in a language independent manner. Examples will be taken primarily from the Smalltalk-80 (TM of ParcPlace Systems) programming system, but the reader need only refer to some of the other articles in this issue of Computer Music Journal for descriptions of systems

1. stp@create.ucsb.edu

based on other languages and programming environments. No basic introduction to the terms or techniques of object-oriented languages will be presented here. The introductory materials found in any of (Krasner 1980), (Lieberman 1982), (Flurry 1989), (Peterson, 1987), (Byte 1981), (Byte 1986), (Goldberg and Robson 1983), (Goldberg 1984), (Stefik and Bobrow 1986) or (Meyer 1987) are recommended to readers not familiar with object-oriented programming concepts and terminology.

Factors that make up Object-Oriented Programming

According to the current lack of unity as to the definition of object-oriented programming, the most fitting initial definition should be a list of the features that describe existing artifacts that are acknowledged as being "object-oriented." This list of features can be factored into several groups related to language issues, problem-solving and design features, types and interpretations of inheritance and reuse, programming environments, and user interface management systems.

The language-related characteristics of common object-oriented systems include explicit separate description of the state and behavior of objects; interaction of objects via (asynchronous or concurrent) message-passing; strict (semantically-rooted) encapsulation and modularity between objects; distinct external and internal views of abstract data structures; and late-binding of message name to method body by the receiver of a message.

The aspects of these systems that effect the problem-solving, analysis and design phases are the description of module interfaces that describe the "what" rather than the "how"; standardized class interface description format (class lattices and diagrams); design for maximal reuse and sharing of interfaces and implementations; rapid prototype implementations and multi-step incremental redesign; delegation and forwarding of protocols or roles among objects; and the viewing of protocol families as roles or local sub-languages.

The reuse of modules via inheritance in object-oriented systems has many criteria including class/instance-based versus prototype-based inheritance; inheritance via composition, refinement, factorization and abstraction in class hierarchies; abstract data types representing aspects or roles in domain modeling; single or multiple inheritance hierarchies (with trees or lattices as models); and the availability of flexible control structures based on the reification of blocks or closures as first-class objects.

Many people equate object-oriented programming with integrated programming environments. Some of the characteristics of integrated object-oriented systems are support for interaction with objects and browsing/navigating within hierarchies of classes and instances; interactive inspectors and editors for manipulating objects; instance catalogs and persistent collections; rapid development cycle turnaround due to encapsulation and late-binding; computer-assisted software engineering (CASE) support

with object-oriented notations, processes and tools; and special support tools for re-design and -implementation.

The higher-level user interface-related features of a dynamic object-oriented user interface management system include its support for kits of pluggable user interface components and interactive view construction tools; flexible object-oriented interaction paradigms, input modeling via control delegation, selection agents, continuation-passing or event filters; and integrated domain-specific frameworks and highly-customizable applications.

Object-Oriented Software Design Concepts

Adele Goldberg (known as one of the pioneers of the Smalltalk systems at the Xerox Palo Alto Research Center in the 1970's and 1980's), drew up a list of four steps to learning object-oriented design and programming at a recent panel discussion on teaching object-oriented programming. She classifies these steps as: processes of decomposition and composition; recognition of similarities and differences; behavioral description of kinds of objects; and the application of metrics and examples of "good design."

The central role of decomposition and composition (related to analysis and synthesis) in the design of software systems is shared with several other disciplines in which complex systems are being constructed (e.g., musical *composition*). The design of modeling hierarchies involves the analysis of a set of objects in terms of the similarities and differences of their properties and behaviors. The description of objects as combinations of state and behavior permits an increased level of reuse through the sharing of protocol among classes with different state variables and different implementations of the shared protocol.

The sharing of message protocol that is found within most abstract/concrete class hierarchies (which are also known as *protocol families*) is evidenced by the fact that users tend to group these families in terms of their common protocol. For instance, one thinks of Streams as objects that can be accessed via messages such as *next* and *nextPut:* for reading or writing to them, or of DisplayObjects as objects that can be sent *display* messages (note that we follow the Smalltalk style and capitalize the names of class). The most generally-cited among the many metrics of quality for object-oriented software has become the level of reuse or sharing. This is interesting largely because it runs counter to many traditional metrics of productivity; a programmer's productivity has often been measured by the amount of code produced, but now 'less is better' and small hierarchies with *power through reuse* are more highly valued.

Levels and Types of Reuse

There is already a rich literature on software reuse and techniques for improving reusability. Several special works therein relate to reuse in object-oriented systems and object-oriented design for maximal reuse; see especially (Pope, Goldberg and Deutsch

1987), (Deutsch 1983), (Freeman 1986), and (Johnson and Foote 1988). For the description of object-oriented systems, there are generally four types of reuse that are recognized and differentiated:

- reuse of algorithms across data structures, a form of data abstraction allowing algorithms to be shared by different data structures;
- reuse of module interfaces, a system designed to be portable as a whole to a variety of machines with modest effort, allowing reuse of the complete system across hardware environments;
- reuse of frameworks across applications, many interchangeable building blocks and conventions for implementing interactive applications; and
- reuse of tools across languages, programming tools that can be re-applied to programming in other languages.

Object-oriented systems are designed to facilitate reuse of both code and design. This reuse can take place at granularity levels ranging from individual algorithms and data structures to entire systems, and can provide sharing across a range of both hardware (which supports the system from 'below'), and applications (which employ the system from 'above'). The two types of reuse of design can be labeled *reuse of specification* (sharing of protocols or roles), and *reuse of implementation* (inheritance or refinement of algorithms).

As an aside, we note that the ability of a language and a system to support reusability is meaningful only if the programmer can: (1) find the objects to be reused and (2) determine how to reuse them. This changes the software development process to be one of searching for information and reading about existing capability (Goldberg 1986).

Object-Oriented Design Techniques

The process of software design and implementation is often divided up into several phases, and one often hears of special techniques and tools for these phases. The development process most closely associated with object-oriented design is that of rapid prototyping and incremental re-design. The use of object-oriented languages' abstract data types and of the support for interaction found in many object-oriented programming environments also encourages phases of exploratory programming ((Alexander 1985) and (Diederich and Milton 1987)).

Many techniques for structured software design identify the phases of *analysis* and *synthesis*, whereby the analysis phase is often mapped onto the process of top-down functional decomposition. Object-oriented modeling techniques, on the other hand, encourage multi-stage implementation and incremental re-design rather than multi-stage design process preceding prototype implementation. The proponents of object-oriented design identify programming cases where no *a priori* decomposition or top-down design can be drawn up (i.e., "there is no top"). Other cases are cited where the abstractions that

will allow maximal reuse and sharing cannot be generated from the specification, but can result from rapid prototyping and incremental re-design.

We will present four central object-oriented design techniques—composition, refinement, factorization and abstraction—in the following sections. We describe them as belonging to two passes, the prototype pass and the second (or third or n-th) generation design and implementation refinement. Composition and refinement are the two most prevalent pass one techniques. We will define and illustrate each with examples.

Composition

Composition involves the reuse of existing classes by using them as instance variables of new classes. The new composed class then forwards (delegates) protocol to its instance variables; the composed class is a client of the instance variable's external protocol. The classes involved are related by a *has-a* relationship; e.g.,

composedClass *has-a* instanceVariableClass.

This can also be seen as:

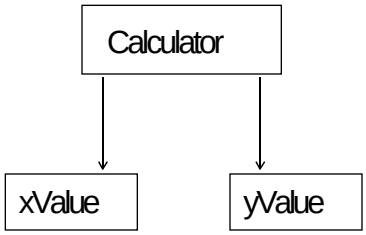
composedClass *delegates-to* instanceVariableClass.

The reuse of entire sub-protocols of the instance variable's class is common; one speaks of the 'mixin' of roles (e.g., Stream-ness or Magnitude-ness).

In our first example, we attack the task of designing a simple calculator model capable of responding to messages such as:

```
calc := Calculator new.
calc  x: 1.34.
calc  y: 5.45.
calc  multiplyXtimesY.
```

The design technique we'll use is the composition of two Magnitudes into a calculator class whose instances understand the simple calculator language as their protocol. The steps are to define a model class whose state includes x and y values and which has protocol for accessing and operating on them. Figure 1 illustrates the composition of the calculator model and shows its class diagram. In the class diagram, we can see the class and its superclass, the class's instance variables, and the protocol of the class (methods it implements). Comments describing the behavior of the class's methods are shown within double-quotes in the class diagram.



(links represent 'has-a' relationship)

Calculator Composition Diagram

name	**Calculator**
superclass	Model
instance variables	x "a magnitude or number" y "a magnitude or number"
accessing methods	getX "answer the x value" setX: "set the x value" getY "answer the y value" setY: "set the y value"
operations	addYtoX "add the two into X" subtractYfromX multiplyYtimesX divideXbyY

Calculator Class Diagram

Figure 1: Composition Example: Calculator Class

A musical example of composition would be to start to implement a model of EventLists for simple music description formats such as:

```
aNote := Event duration: (1/4 beat)
     pitch: 'c#3'
     voice: 'violin'
     loudness: 'mezzo-forte'
     dynamic: 'con legno'.
anEventList ¨ EventList new.
eventList add:  (aNote)
     at:  (eventStartTime).
   .  .  .
eventList add:  (aNoteEventDescription)
     at:  (eventStartTime).
eventList play.
```

(see the Music-N-type languages, (Buxton 1984), (Dyer 1984) or (Pope 1986)).

The composition of an OrderedCollection and an Event will serve quite nicely here. One simply defines an Event subclass that includes a Collection of events. It will inherit Event state and protocol, and add collection state and protocol for sub-events. It can then reuse the enumeration protocol of collections by forwarding it to the event-collection instance variable. Figure 2 shows the class diagram for the EventList class. It shows EventLists's event protocol (e.g., *play* and *duration*) as well as its collection protocol (e.g., *add:at:* or *select:*).

name	**EventList**
superclass	Event
instance variables	events "a collection of Events"
accessing methods	events "answer the event collection" add: "add a new event to the end" add: at: "add an event at the given time" remove: "remove the argument from the list"
Event methods	duration "answer my duration" playAt: "play my events"
Collection control structure operations	do: "iterate over the events" select: "select from the events" detect: "detect from the events" reject: "reject from the events"

Figure 2: Composition Example: EventList Class Diagram

The examples of composition have shown it to be a familiar technique for describing and constructing interesting behaviors. The facet of the technique that is most similar to structured programming is the delegation of forwarding of messages (or entire protocols) to one's instance variables.

Refinement

The technique of refinement allows one to reuse existing system classes and specialize them via subclassing. The subclass may extend the inherited state (add instance variables) and / or extend or refine the inherited behavior (add or override methods). The new subclass can see and / or override the inherited class's internal state and method implementations. The two classes involved in refinement subclassing are linked by the relationship of:

subclass *is-a* superclass

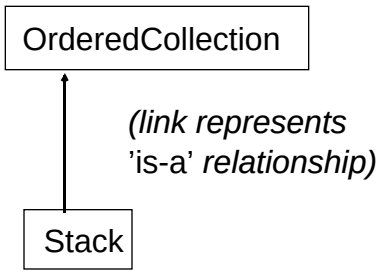
or

subclass *inherits-from* superclass.

The direct inheritance of method implementations allows overriding of specific methods or protocols (e.g., re-implementation of private protocol).

The simplest example of refinement we will present is building a simple push / pop stack via refinement of the state and behavior of OrderedCollections. This can be accomplished by defining a new OrderedCollection that inherits its indexable instance variables and accessing protocol such as *first* and *addFirst:*. We then refine the implementation of the inherited collection accessing methods providing for different

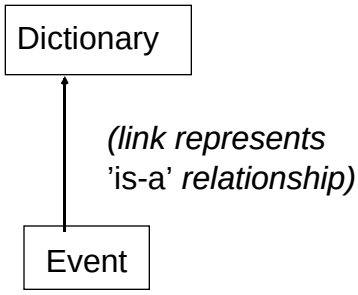
over- and underflow behavior, and rename them to be *pop* and *push:*. Figure 3 shows the refinement structure and the class diagram for the Stack class.



name	**Stack**
superclass	OrderedCollection
instance variables	(none)
accessing methods	push: "uses inherited addFirst: method" pop "uses inherited removeFirst method" isEmpty "inherited" peek "uses inherited first method"

Figure 3: Refinement Example: Stack Class Diagram

A musical example of refinement is the design of a flexible framework for music description and processing using event and event-list models (see the EventList example above). The technique of refinement is applied here to make a new subclass of Dictionary that adds a new named instance variable (duration) and flexible property-list protocol. We define a new Dictionary (property list) subclass which inherits Dictionary's protocol for property list behavior and adds one state variable and new protocol for accessing and scheduling. In Figure 4 one can see the class diagram for Events; note the different protocols it demonstrates.



name	**Event**
superclass	Dictionary
instance variables	duration "my duration time"
accessing methods	duration "answer the value" duration: "set the value"
dictionary operations	at: "answer a property's value" at: put: "set a property's value"
playing	play "play me now" playAt: "play me at the given time"

Figure 4: Refinement Example: Event Class Diagram

Refinement is the typical inheritance-based design technique. The reuse and refinement of inherited state and behavior involves extension of inherited state and protocol and / or customization (through overriding), of inherited method implementations.

The later pass techniques are centered around processes that transform the descriptions of class hierarchies in order to maximize reuse and/or sharing. These two techniques are known as factorization and abstraction, and are illustrated next in several examples.

Factorization

The factorization technique is applied in the case where one wants to increase the reusability by describing a complex class in terms of a hierarchy of possibly concrete aspects. The process of factorization involves reifying the roles or aspects of a class into several levels of more reusable classes; one constructs a hierarchy of refinement and composition out of a single class. The newly-defined classes normally represent abstractions from the domain of discourse that provide further opportunities for reuse. Factorization emphasizes sharing of state and accessing behavior; it leads to abstract classes less likely to have complex behavior, and to vertical hierarchies of "mixins" for further composition or refinement steps

The train example describes the design of a model of a control system for a passenger train using the techniques of rapid prototype implementation and redesign via factorization into a 1-Dimensional hierarchy. The first pass (shown in Figure 5), consists of building a monolithic class called Train with state such as capacity, passenger count, location and velocity. In the second pass, the large and complex Train class becomes a framework for reuse, and we then split it into a hierarchy of Container (with state for capacity and contents), Vehicle (with state for location and velocity), and Train (which adds no state but overrides some inherited method implementations). The increased abstraction allows easier extension of the framework via refinement of the intermediate classes, e.g., adding other Containers (such as coffee cups) or Vehicles (such as cars). Figure 5 shows the before and after class diagrams for the Train example. In the first class diagram, we see the single-class solution. The second diagram shows the factorized version as a single-strand inheritance hierarchy.

name	**Train**
superclass	Model
instance variables	capacity passengers location velocity
accessing methods	capacity: "set capacity addPassenger: "register a passenger" passengerWeight "answer the sum" location "answer location" velocity "answer velocity" velocity: "set the velocity"
operations	checkCapacity "make sure you can go" move: "change location"

First-pass implementation

(links represent 'is-a' *relationship)*

name	**Container**
superclass	Model
instance variables	capacity contents
accessing methods	(...)
operations	(...)

name	**Vehicle**
superclass	Container
instance variables	location velocity
accessing methods	(...)
operations	(...)

name	**Train**
superclass	Vehicle
instance variables	(none)
	(accessing methods inherited)
operations	(Train-specific methods)

Second-pass re-design

Figure 5: Factorization Example: Two Passes at Trains

A musical example that demonstrates factorization well is the design of a framework for rapid implementation of interactive user interfaces for musical EventList editing. We again choose to do a rapid prototype implementation and then redesign via factorization into a rich multi-dimensional hierarchy. The first step is to design a catch-all class for event list editing. The EventListEditor class will include methods for displaying staves and event lists in some sort of music notation; for selecting, grouping and operating on events; for scrolling and zooming; and for reading and writing event lists to and from disk files. The later passes will use factorization to split this complex and non-reusable class into a family of related classes each of which will model some important part of the functionality. The DisplayList and DisplayList LayoutManager classes form the basis of the graphical presentation kernel. Instances of EventListEditorController, ScrollManager and the SelectionManager are the central interaction objects, and the various function-related managers (e.g., StoreManager) are used for the delegating the many peripheral operations. By re-factoring the EventListEditor, we achieve greater readability and flexibility, and we provide opportunities for increased reuse by decomposing the functionality of the editor into smaller, more domain-related chunks. Figure 6 shows the two EventListEditor class diagrams.

name	**EventListEditorView**
superclass	View
instance variables	model layoutProperties controller scrollingProperties "lots more state"
accessing methods	verticalSpacing staffSize "lots more"
operations	hitDetectModelForPoint: scrollViewBy: "lots of misc. operations"
displaying	displayView generateCachedView "lots of code in here"

First-pass implementation

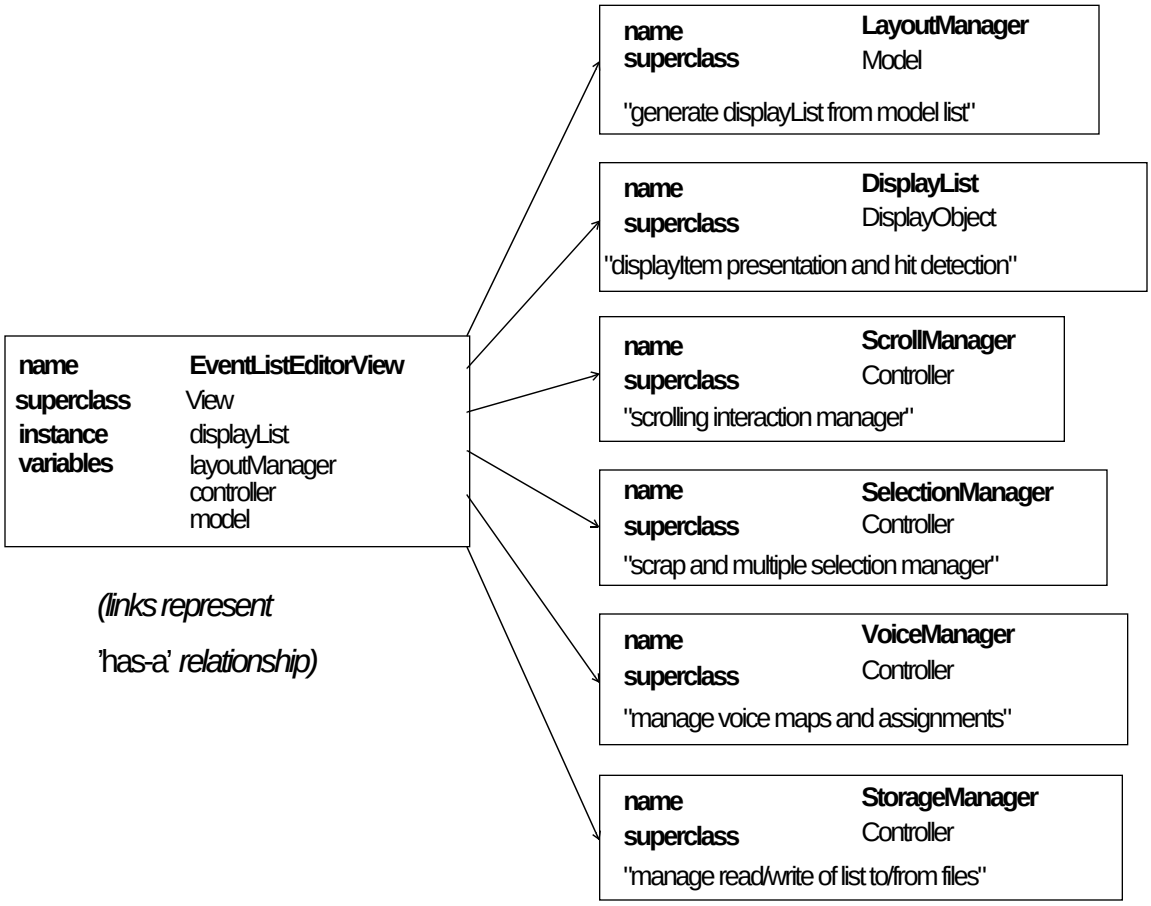


Figure 6: Factorization Example: EventListEditor Re-factoring for reuse and pluggability

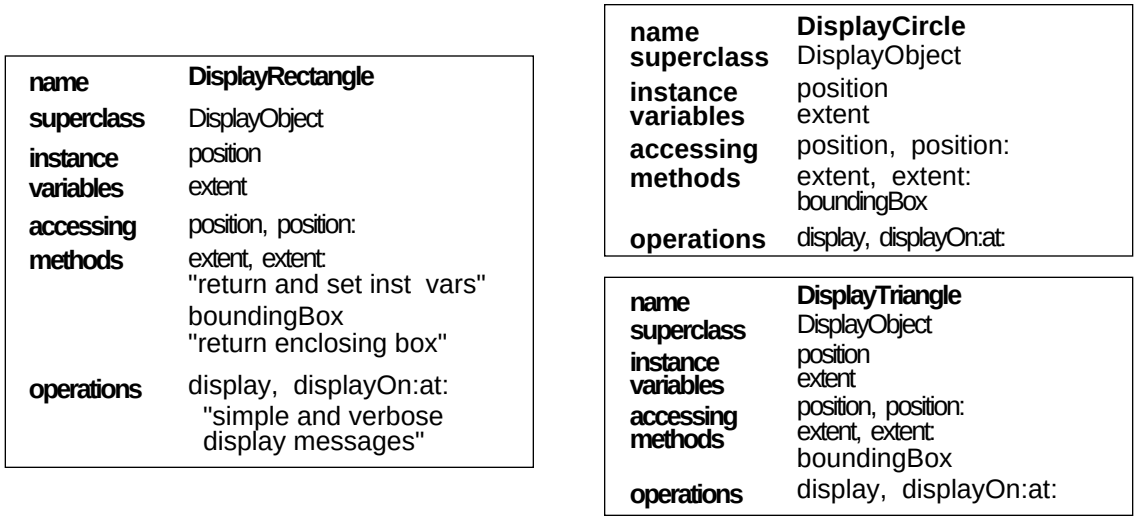
We see that factorization can lead to increased reuse via composition or refinement in hierarchies that start out as "large objects" whose state can be factored, or as classes with complex protocols that can be split off into aspects or roles.

Abstraction

The design technique of abstraction can be used to increase the amount of reuse within a set of classes by describing abstract classes that embody some of their shared state and / or behavior. One uses abstraction to construct a hierarchy relating several concrete classes to common abstract classes for better reuse of state and behavior. This emphasizes sharing of both state and behavior; abstract classes often implement algorithms in terms of methods that are overridden by concrete subclasses. The result of abstraction is a more horizontal hierarchy or a new protocol family.

A typical abstraction example is building a model graphical objects for an object-oriented drawing package. The first pass includes several concrete classes (for the

different types of displayable objects). The later passes will the use abstraction to re-implement these classes as a hierarchy with significant sharing among abstract and concrete classes. One would start by prototyping a wide and shallow hierarchy of DisplayItems from a collection of classes with little reuse, the first pass implements DisplayRectangle, DisplayCircle and DisplayTriangle with common protocol but no inheritance among them and no state or behavior sharing. The second pass adds abstract class DisplayItem with position, size and orientation instance variables and protocol for manipulation and display, concrete subclasses now inherit all their state and reuse shared protocol specification and implementations of most algorithms. The third pass further refines this, adding another level of abstract modeling for line-segment- and arc-based DisplayItems; and easy extensibility is achieved. Further passes might identify Square as a special case of Rectangle and Circle as a special case of EllipticalArc. Figures 7 and 8 show three of the possible class hierarchies for DisplayItems.



First-pass implementation

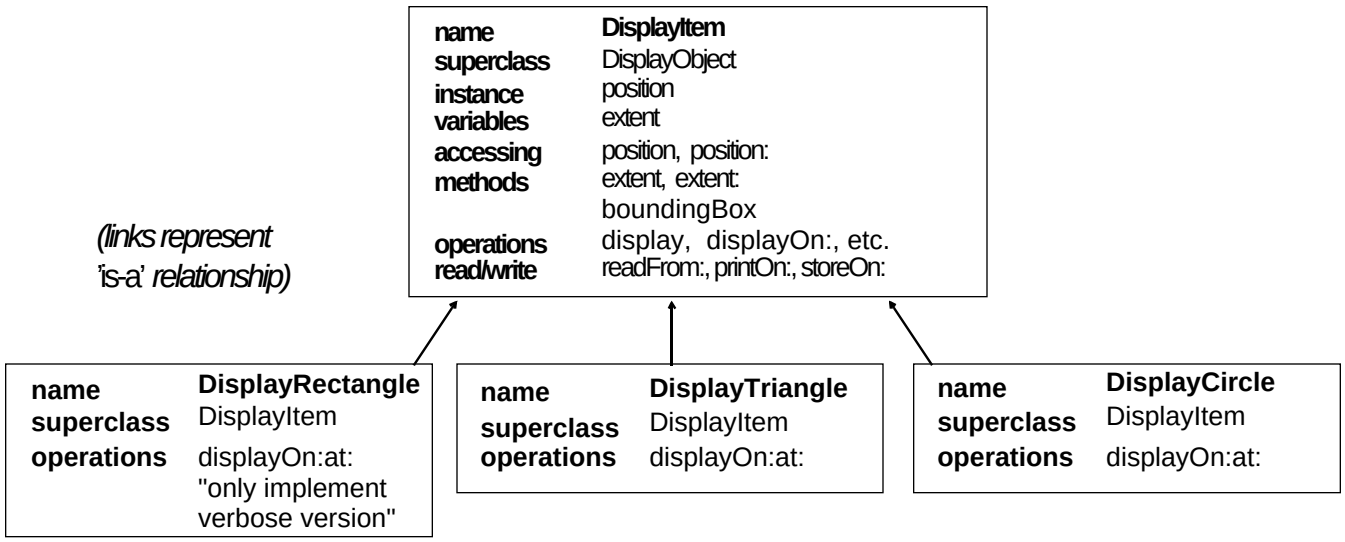


Figure 7: Abstraction Example: Adding Abstract DisplayItem Classes

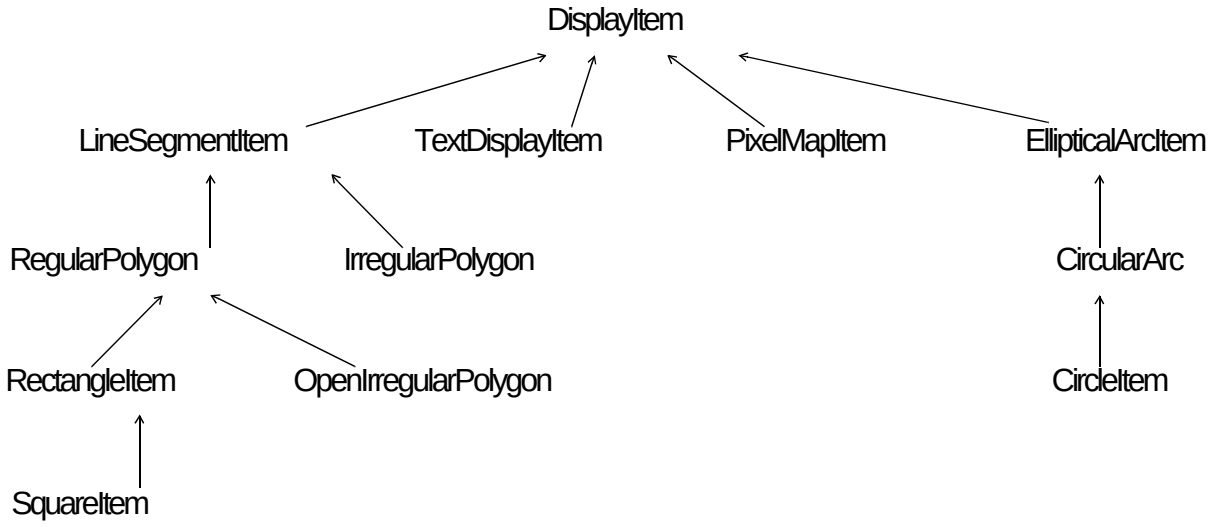
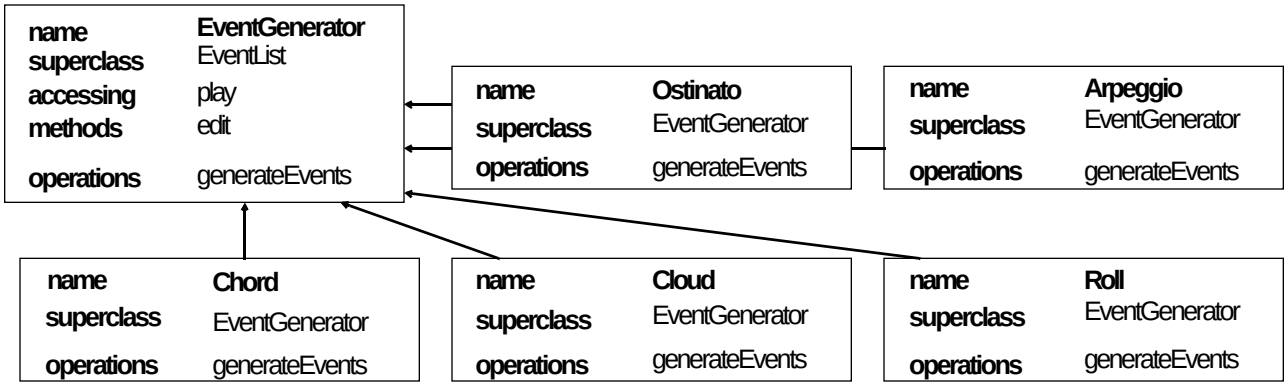


Figure 8: Abstraction Example: Possible Refined DisplayItem Hierarchy

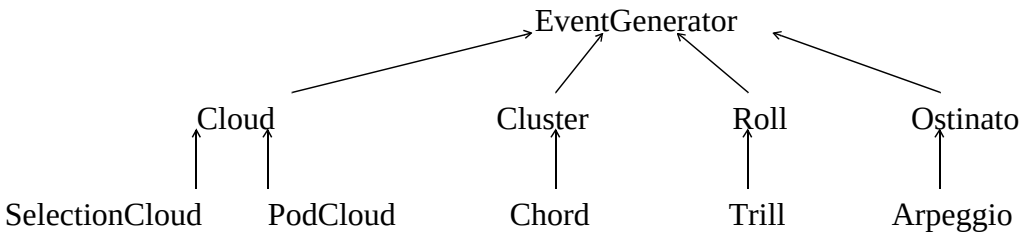
There are many approaches to the architecture of a framework for algorithmic composition or an algorithmic music description language. We will illustrate the design as a multi-step process that starts with the construction of an abstract class named EventGenerator as a subclass of EventList. Subclasses of EventGenerator should implement the message *generateEvents*, so we define it in the abstract class as

"subclassResponsibility" (i.e., it is expected that subclasses will override it). The first pass design's concrete classes are Chord (which might hold its root and its inversion), Roll (which has a single event and can repeat it), Trill (repeat an eventList), Arpeggio (play with inter-note skew), Ostinato (repeat as a process), SelectorCloud (select from given pitch, amplitude and voice sets) and PodCloud (select from given ranges). They share little state except what EventGenerator's inherits by being a subclasses of EventList. Several of them add instance variables to deal with their particular representations; and each implements its own *generateEvents* method.

Shared state and behavior among EventGenerators is often not apparent until after the first implementation; and in the implementation of further subclasses, we use some of the first pass classes as abstract intermediate classes. We try to push as much of the method implementations as possible into the abstract class EventGenerator; protocol and method implementations for instance creation, accessing and user interface can now be shared. Increased abstraction inserts abstract classes Cluster (a more general notion for chords), Cloud (an abstract stochastic EventGenerator), and makes Trill a subclass of Roll (that repeats a collection of notes rather than just one). In figure 9 one can see the first pass class diagram for EventGenerators and one of the possible abstraction-based refinements of this design.



First-pass implementation



Second-pass re-design

Figure 9: Abstraction Example: Intermediate Abstract EventGenerator Classes

There are several trade-offs in using abstraction-based techniques. The DisplayItem and EventGenerator examples could each be used to demonstrate extremes of factorization or abstraction. Examining the construction of classes for the generation of many types of Chords, for example, one might implement a prototype that includes a class called Chord with many different methods for generating EventLists according to given rules and root pitches. The prototype might be cleaned up so that the types of chords (e.g., major, minor, 4th, pentatonic) are grouped into protocols in class Chord. The extreme of abstraction would be to have a deep and wide hierarchy of Cluster subclasses (e.g., classes like MajorEleventh, SixteAjoute, PentatonicCluster, FourthRow, or CParker1957), each with one method (e.g., *generateEvents*). It is not always possible to say which of these solutions is better. The design that implements a single, very large Chord class (in terms of number of methods) will probably lead to a more readable system and a simpler architecture, but will have almost no reuse. The alternative implementation, with a large number of one-method classes, might be more easily extensible, but possibly also more difficult to navigate within.

Multi-Step Techniques for Design and Implementation

The relationship between rapid prototyping and design has become muddied in this discussion. This is intentional and is a by-product of object-oriented design. The prototyping phase is included in several of the object-oriented design techniques because it is assumed that it is not always possible to arrive at a class hierarchy that maximizes sharing and reuse in the first pass.

The object-oriented design literature certainly includes numerous proponents of other sets of beliefs, models, terms and methodologies than the one presented in this discussion. The other software architecture articles in this issue of Computer Music Journal present some similar and some alternative approaches, as do the highly-developed design methodologies used in the ADA programming language by Grady

Booch (Booch 1986), in Eiffel by Bertrand Meyer (Meyer 1987), and in LISP/LOOPS by Mark Stefik and Daniel Bobrow (Bobrow and Stefik 1984). More basic design methodology references such as (Cardelli and Wenger 1985), (Meyer 1986), or (Shlaer and Mellor 1988) can be recommended for further study.

Application Development in an Integrated Object-Oriented Programming Environment

The most well-known integrated interactive object-oriented programming systems are Smalltalk-80 (Goldberg 1984), LispMachine LISP (Roads 1983), and INTERLISP-D (Tietelman and Masinter 1981). There are several general characteristics brought about by the mixture of a high-level integrated programming environment—along the lines of (Barstow, Shrobe and Sandewall 1984) or (Deutsch and Taft 1980)—with one of these flexible interactive object-oriented programming systems. In a sense, program development by rapid prototyping and incremental refinement becomes so strongly the order of the day that the programming environment and tools themselves are malleable open applications constructed as hierarchies for reuse by the techniques inherent in the host language. In general, these systems have promoted dynamic, multi-phase incremental prototyping and development through the common steps of the design of abstract classes and/or reuse of existing system classes, of subclassing abstract classes to construct concrete classes, and of defining user interfaces by subclassing for new behaviors and "plugging in" existing components where appropriate.

One can group object-oriented software packages by the characterization of their external interfaces and their primary modes of reuse into three categories. We call these: *frameworks*, *toolkit*s, and *customizable applications.*

Frameworks

The examples of frameworks that were presented above used refinement and composition of existing classes as first-pass techniques and abstraction or factorization in the second and further refinements to produce hierarchies of abstract and concrete classes that used common protocols and defined new shared behavior. The Collection and Stream classes are two of the distinguished protocol families (i.e., frameworks) within the Smalltalk-80 systems. They each define state variables and behavior methods at several levels of abstraction and refinement. The client-side view of frameworks is often that the user uses or overrides the implementation of a few of the important methods, so the refinement-oriented user is on the inside of these frameworks,seeing the implementation of their methods.

The Event, EventList and EventGenerator classes discussed in the examples represent a framework that is easily extensible by further refinement describing new types of behavior. Several beautiful framework architectures are presented in the other articles in this issue of Computer Music Journal. Kyma's sound model classes, the VDSP virtual

digital signal processing models of data types and processors, Ttrees, or the CreationStation's schedules can serve as examples.

Toolkits

In the examples of the use of toolkits, we saw the incorporation of existing code by direct reference or by composition and delegation. The elements of toolkits tend to be reused by composition, where they incorporate their behavior into the behavior of the aggregate. The client-side view of toolkits is (hopefully very simple) message interface, such as eventGenerator or pluggableView instance creation messages. Another view is that the elements of a tool kit all are members of the same protocol family and share some small interface, e.g., those same view creation messages.

The reuse of user interface components is common in integrated object-oriented systems, where applications are often modeled as model objects being displayed in view objects and being manipulated by controller objects. The view and controller components are taken from standard libraries that might include different types of screen presentation views and interaction monitors. Examples of toolkits of pluggable components for music user interfaces are the NeXT SoundKit and MusicKit (TM of NeXT, Inc.) discussed in this issue of Computer Music Journal, and my own HyperScore ToolKit's pluggable user interface components and event list editors (Pope, 1987).

Customizable Applications

The notion of an open application, where a high degree of end-user customization is supported, is not novel for software development tools. A significant number of object-oriented applications (especially in music-related areas), attempt to extend this openness into the range of non-programming-oriented domains.

Kyma, Javelina, Forest and the HyperScore ToolKit are among the musical applications that are built as end-user application tools built into reusable environments with frameworks and / or toolkits for user customization and extension.

Conclusions

The five most currently-pressing problems in software applications development are software complexity, portability, development effort, user interface flexibility and maintainability. It is the goal of object-oriented software design to address these areas by providing support for the process of rapid prototyping and incremental design, and improved reuse and sharing.

We have described object-oriented problem-solving and analysis in terms of modeling of abstract / concrete description structures, and have discussed the object-oriented design methodology. In several examples, the problem-solving techniques of decomposition / composition, behavioral analysis and the development of special-purpose protocols or languages were presented, and four specific techniques that can be

applied during a multi-step specification / design / implementation cycle were demonstrated. The use of exploratory programming techniques can be supported in programming environments through integrated systems with very fine-grain incremental development and high levels of interactivity.

The analysis of, and design for, reuse of several kinds on several levels is central to many of these techniques, and is a theme running through the other articles in this issue of Computer Music Journal.

Acknowledgments

Many of the ideas expressed in this article come from discussions the author had with L. Peter Deutsch, Adele Goldberg, Glenn Krasner and Kenny Rubin. Adele Goldberg, Kenny Rubin, Glenn Krasner and Cliff Hollander also read early versions of this article and provided valuable comments.

References

Alexander, James H. 1985. *Exploratory Programming Using Smalltalk*. Beaverton: Computer Research Laboratory, Textronix Inc.

Barstow, David, H. Shrobe and E. Sandewall. 1984. *Interactive Programming Environments*. New York: McGraw-Hill

Bobrow, Daniel and Mark Stefik. 1984. *The LOOPS Manual*. Palo Alto: Xerox Palo Alto Research Center

Booch, Grady. 1986. *Software Engineering with ADA*." Menlo Park: Benjamin / Cummings Publishers

Buxton, William et al. 1978, *The Structured Sound Synthesis Project (SSSP): An Introductio*n. Technical Report CSRG-92, University of Toronto

Byte Magazine. 1981. Special Smalltalk-80 Issue. *Byte, The Small Systems Journal,* 6(8) August, 1981

Byte Magazine. 1986. Special Object-Oriented Programming Issue. *Byte, The Small Systems Journal,* 11(8) August, 1986

Cardelli, Luca and Peter Wenger. 1985. "On Understanding Types, Data Abstraction and Polymorphism." *ACM Computing Surveys* 17(4)

Deutsch, L .Peter and E. Taft. 1980. *Requirements for an Experimental Programming Environment*. Palo Alto: Xerox Palo Alto Research Center report CSL-80-10

Deutsch, L. Peter. 1983 "Reusability in the Smalltalk-80 Programming System." *Proceedings of the ITT 1983 Workshop on Reusability Programming*. reprinted in (Freeman, 1987).

Diederich, Jim and Jack Milton. 1987. "Experimental Prototyping in Smalltalk." *IEEE Software,* May 1987 pp. 50-64.

Dyer, Lounette M. 1984. "Toward a Device Independent Representation of Music." *Proceedings of the 1984 International Computer Music Conference*. pp. 251-256. San Francisco: Computer Music Association.

Flurry, Henry. 1989. "An Introduction to the CreationStation and its Design." *Computer Music Journal* 13(2): XX-XX

Freeman, Peter, ed. 1987. *Tutorial: Software Reusability*. IEEE Press, 1987

Goldberg, Adele and David Robson. 1983. "*Smalltalk-80, The Language and its Implementation*." Reading: Addison-Wesley (also available in Japanese)

Goldberg, Adele. 1984. "*Smalltalk-80, The Interactive Programming Environment*." Reading: Addison-Wesley

Goldberg, Adele. 1986. "Programmer as Reader." *Proceedings of the IFIP Information Processing Conference 1986*. Kugler, J. H., ed. Amsterdam: Elsevier Publishers

Johnson, Ralph and Brian Foote. 1988. "Designing Reusable Classes." *Journal of Object-Oriented Programming* 1(2): 22-35

Knudsen, Joergen and Ole Madsen. 1988 "Teaching Object-Oriented Programming is more than Teaching Object-Oriented Programming Languages." *Proceedings of the Second European Object-Oriented Programming Conference.* pp. 21-40. Berlin: Springer Verlag

Krasner, Glenn. 1980. "Machine Tongues VIII: The Design of a Smalltalk Music System." *Computer Music Journal* 4(4): 4-22

Lieberman, Henry. 1982. "Machine Tongues IX: Object-Oriented Programming." *Computer Music Journal* 6(3): 8-21

Madsen, Ole and Birger Moeller-Pedersen. 1988. "What Object-Oriented Programming May Be—and What it Does Not Have to Be." *Proceedings of the Second European Object-Oriented Programming Conference*. pp. 1-20. Berlin: Springer Verlag

Meyer, Bertrand. 1988 *Object-Oriented Software Construction*. New York: Prentice-Hall

Peterson, Gerald. E. ed. 1987. *Tutorial: Object-Oriented Computing* (in two volumes). New York: IEEE Press

Pope, Stephen T. 1987. "A Smalltalk-80-based Music Toolkit", *Proceedings of the 1987 International Computer Music Conference*. pp. 166-173. San Francisco: Computer Music Association. excerpted in *Journal of Object-Oriented Programming* 1(1): 6-11

Pope, Stephen T., Adele Goldberg and L. Peter Deutsch. 1987. "Object-Oriented Approaches to the Software Lifecycle Using the Smalltalk-80 System as a CASE Toolkit". *Proceedings of the 1987 IEEE/ACM Fall Joint Computer Conference* NewYork: ACM Press. pp. 13-20

Roads, Curtis. 1983. "*Symbolics 3600 Technical Summary*" Cambridge: Symbolics, Inc.

Shlaer, Sally, and Stephen J. Mellor. 1988 "*Object-Oriented Systems Analysis: Modeling the World in Data*" New York: Yourdon Press / Prentice-Hall

Shriver, Bruce and Peter Wegner, eds. 1987. "*Research Directions in Object-Oriented Programming*." Cambridge: MIT Press.

Stefik, Mark and Daniel Bobrow. 1986. "Object-Oriented Programming: Themes and Variations." *AI Magazine* 6(4): 40-62. Menlo Park: AAAI

Stroustrup, Bjorn. 1987 "What is Object-Orented Programming?" *Proceedings of the 1987 European Object-Oriented Programming Conference*. Paris: AFCET

Tietelman, Warren and Larry Masinter. 1981. "The INTERLISP Programming Environment." *IEEE Computer* 14(4)

Wegner, Peter. 1987. "Dimensions of Object-based Language Design." *Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages and Applications* pp. 168-182. New York: ACM Press