Using Animation to Design, Document and Trace Object-Oriented Systems

Technical Report GIT-GVU-92-12

John J. Shilling
John T. Stasko

Graphics, Visualization, and Usability Center
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

E–mail: {shilling, stasko}@cc.gatech.edu

Abstract

Current diagramming techniques for the development and documentation of object-oriented designs largely emphasize capturing relationships among classes. Such techniques cannot capture full designs because the static nature of class relationships makes them inadequate for describing the dynamics of object collaboration. Other techniques attempt to diagram dynamic behavior but are limited by their media to producing essentially passive description of dynamic operations. What is still needed is a technique that models message ordering, changing visibility and temporal object lifetimes in a manner that is concise and immediate. We have developed an approach in which developers use animation to develop and capture object-oriented designs. This allows developers to design object-oriented scenarios in the way that they visualize them: by animating the actions of the objects in the scenario. The same animation then acts as the documentation for the design. Its playback makes immediately evident the temporal relationship of object messages, object creation, object destruction, and changing object visibility. Our technique is supported as part of a suite of object-oriented development tools we call GROOVE.

1 Introduction

A key problem to be addressed in the design of object-oriented systems is how object instances collaborate to implement globally visible behaviors. A naive view of the semantics of an object-oriented system focuses only on the class hierarchy, class visibility and the interfaces of individual classes. These forms of documentation are inadequate by themselves because of their static nature. They do not provide direct information on how a system will actually operate. They merely provide information on how classes in the class hierarchy are formed and how they *may* relate to each other during execution. For self-contained service classes this is sufficient but for frameworks and applications much more information is needed. The use of a *String* object, for instance, may be understood based only upon its interface but even seemingly self-contained classes such as *Set* and *Bag* depend on an equality operator that may be overloaded by the objects they contain. A framework or application can embody a well defined collaboration between a large number of classes. The fundamental problem is that static documentation has difficulties illustrating how instances of classes collaborate at run-time to implement tasks not specific to a single class.

Developers need to understand how their classes will interact with frameworks, and they need to describe how tasks required of their system are going to be implemented as a collaboration among object instances. They need tools for understanding and describing dynamic behavior.

Currently, developers often document how the core tasks of a system are implemented as a collaboration among classes by drawing protocol diagrams. A typical protocol diagram will show a set of object instances with arrows drawn between instances representing messages. The messages are labeled to show the operations invoked and numbered to show order. A narrative accompanies the diagram to explain what is being accomplished in the diagram, to identify when objects are created and destroyed, and to describe changes in object visibility. Such diagramming is an attempt to show dynamic behavior with a static media. Compromise techniques such as message numbering must be used to introduce time. Since any object that is to receive a message must be present in the diagram, it is up to the narrative to describe when objects are created or destroyed and when object instances come into and out of visibility for other object instances. This approach clearly fails to provide the clarity and conciseness required of good documentation.

Booch [Boo91] combines *object diagrams* without explicit ordering with *timing diagrams* which show the ordering of messages in a diagram. Several timing diagrams may be associated with a single object diagram to show how different scenarios may unfold based on the same underlying object structure. The effect of using an object diagram with a particular timing diagram is much the same as the protocol diagram described above. The timing diagram adds a notion of relative time that messages are expected take to complete and time spent processing between message sends. The fact that a single object diagram may have several timing diagrams gives flexibility, but it also means that a user must look at two different pieces of documentation to understand the operation of a protocol.

Rumbaugh, et. al.[RBP+91], introduce *scenarios* and *event traces* into the dynamic modeling component of the Object Modeling Technique. A simple *scenario* describes a sequence of events that will occur in an existing or proposed system. It identifies events and gives them an ordering. An *event trace* gives further structure to the scenario by associating

the events with sender and receiver objects. In the event trace we see the information that was captured above and some of the same limitations. Objects and message ordering are identified but the dynamics of object lifetimes and visibility are not.

An interesting design technique from our point of view is the use of CRC cards as described by Beck and Cunningham [BC89]. In this work designers are encouraged to physically animate a design as they narrate it:

> We encourage learners to pick up the card whose role they are assuming while "executing" a scenario. It is not unusual to see a designer with a card in each hand, waving them about, making a strong identification with the objects while describing their collaboration.

Our work is in this same spirit. We want to give users as natural a way as possible for describing the animation in their designs.

The techniques described above recognize the need for the description of dynamic behavior but are limited by the media on which they must be used. The diagramming techniques of both Booch and Rumbaugh can be extended easily to show object lifetimes. Showing the changes in object visibility is not as direct but may be modeled through separate diagrams or by showing visibility *events*.

The basic premise of this work is that we can do better if we look to a different presentation paradigm. Through the use of animation we seek to activate humans' keen pattern-matching and visual perception systems. Graphical animation is a dense, rich information medium that can convey large amounts of information rapidly in an easy-to-comprehend fashion. The animation tools we have developed are intended to trigger the mental connections and relationships that must be present for understanding.

2 Overview

2.1 Purpose

The suite of tools we are developing helps software engineers design, develop, document, understand, and evaluate the execution of their code through dynamic program visualizations. At the center of our system is a visual design tool called GROOVE (GRaphical Object-Oriented Visualization Environment) that allows programmers to visually specify both the static structure of a program and the run-time dynamics and protocols. Programmers specify system structure and operations through menus and direct manipulation operations. Figure 1 shows a view of this tool being operated. Here, a programmer has laid out classes and instances, along with method invocations defining a protocol. The pull-down menu at the top provides many program design and specification commands such as "Class create," "Instance create," "Function invoke," etc. The visual entities can either be positioned manually for each command or an automatic layout facility can be utilized. All entities subsequently can be selected with the mouse and dragged to new positions.

The system allows design sessions to be stored in script files and replayed later as animations. In this way, we provide a form of dynamic *visual documentation* that transcends

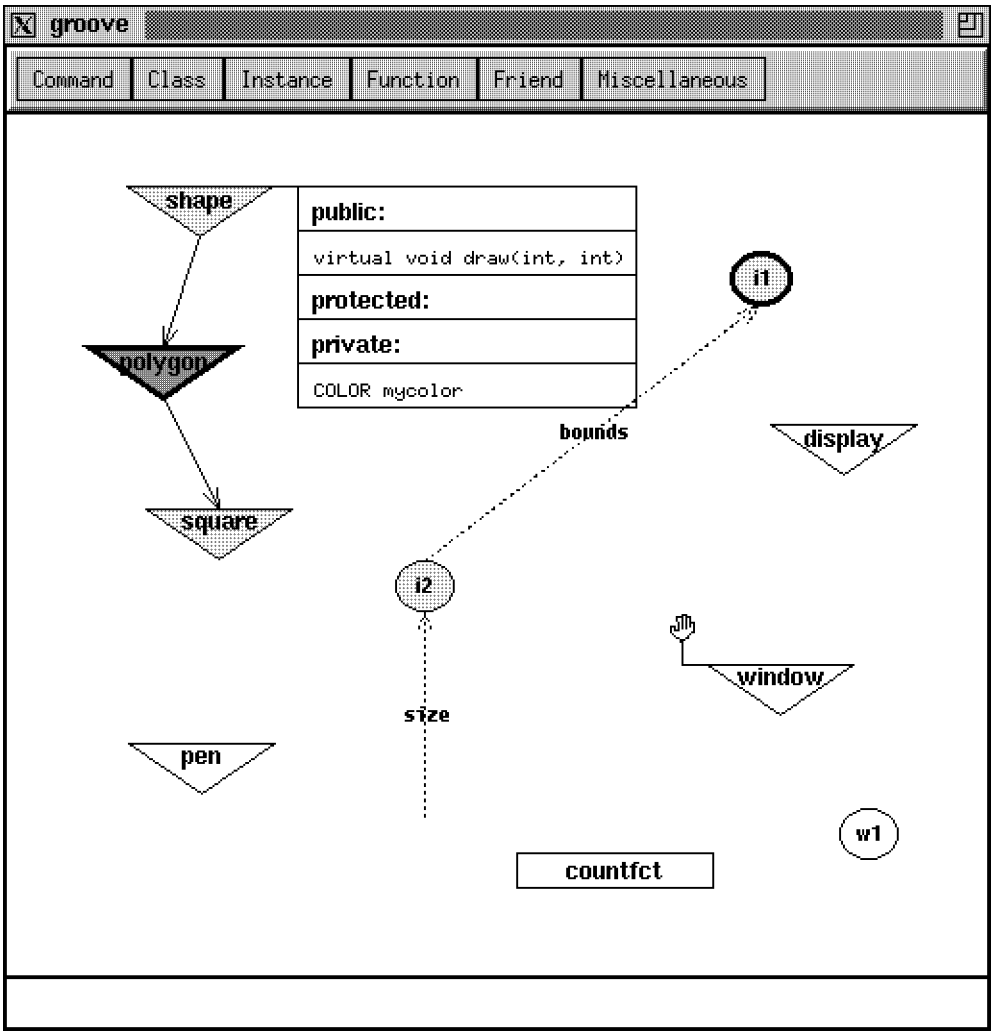

Figure 1: The GROOVE program specification and design editor. Here, a programmer has created classes and instances and has begun to define a message protocol.

static textual or picture explanations. In understanding an object-based protocol or scenario, the sequencing of events is absolutely critical. Because animation shows changes over time, it is better suited for capturing a designer's intentions and work.

As a designer interacts with the visual presentation shown in Figure 1, the system also automatically generates and updates an accompanying code template (shown in a neighboring window) that corresponds to the design. The code can, at any point, be written to a file and used as the basis for subsequent development. This feature, in addition to its practical utility, is a valuable educational aid for students learning object-oriented design. They can focus on the important concepts (with visual reinforcement) and not be hindered by numerous syntactic errors that traditionally accompany learning a new language.

Our suite of tools also supports run-time animation of an executing program. When the system generates the design session code as described above, it also can add visualization hooks to the code, and it can generate a "shadow" version of the file containing graphics

description code. When this shadow file is compiled and linked to the primary code file(s), subsequent program execution activates the graphics views that represent program entities and behaviors. These facilities will be described in more detail later in the paper.

2.2 Visualization Paradigm

All of the tools in our system are based upon a common graphical paradigm which uses a unique combination of shape, color, and animation to portray object-oriented programs and protocols. For instance, the shape of a graphical element is used to depict the type of the program entity being depicted. Classes are represented by "upside-down" isosceles triangles. Arrows from the bottom of a triangle (class) to the top of another class represent the inheritance relationship. Class views can either be compressed (the simple triangle) or expanded, which shows the instance data and methods of the class. In Figure 1 the *shape* class view has been expanded. Instances are represented by circles in the visualization paradigm and functions by flat rectangles. The name of a program entity is centered on its graphical object. To specify messaging, users select with the mouse an object and its method to be invoked. We show an arrow grow out from the current context and strike the target object. On message return, we animate the arrow back to its point of origin.

Our visualization paradigm differs from many others in that it does not concurrently present the relationships between all program entities. When viewing a program, system, or protocol depiction with many objects, instantaneous display of all relationships can quickly lead to an information overload. Just showing the different class hierarchies within a program, perhaps by connected lines and color, might be successful. But when one also tries to concurrently show other relationships such as those between class and instances, instance-to-instance visibility, or language-specific notions such as friendship in C++, a display can quickly become a tangled web of lines, shapes, and colors.

To address this problem, we utilize the concept of a *current focus* to drive the display. Any program entity in the display can become the current focus. Within the program specific tool, this is accomplished simply by selecting the entity's representation with the mouse. When describing a protocol, the object last receiving a message becomes the focus. Once an entity has become the focus, all other objects update their view to reflect their relationship to the focus entity. If an entity has no relationship or link to the focus entity, its view is a simple black outline of its shape surrounding its name. If an entity is somehow related to the focus, we use a combination of visual attributes to depict that relationship.

One of the most natural visual attributes to use in a display is color. We use color to represent the different class hierarchies within a program. The different base classes receive a unique dark color to distinguish them from other classes. As subclasses are derived, their hue remains the same as their base class, but they become lighter (less saturation). Instances also are drawn in the appropriate color to indicate the class from which they originate. We envision presenting classes derived via multiple inheritance with a "zebra-stripe" effect.

We utilize other visual attributes and notations to represent further relationships between program entities. For example, visibility between instances is represented by drawing a larger broken circumference around any instance visible to a current focus instance. Friendship in C++ is shown by drawing an icon of an open hand coming out of a class representation.

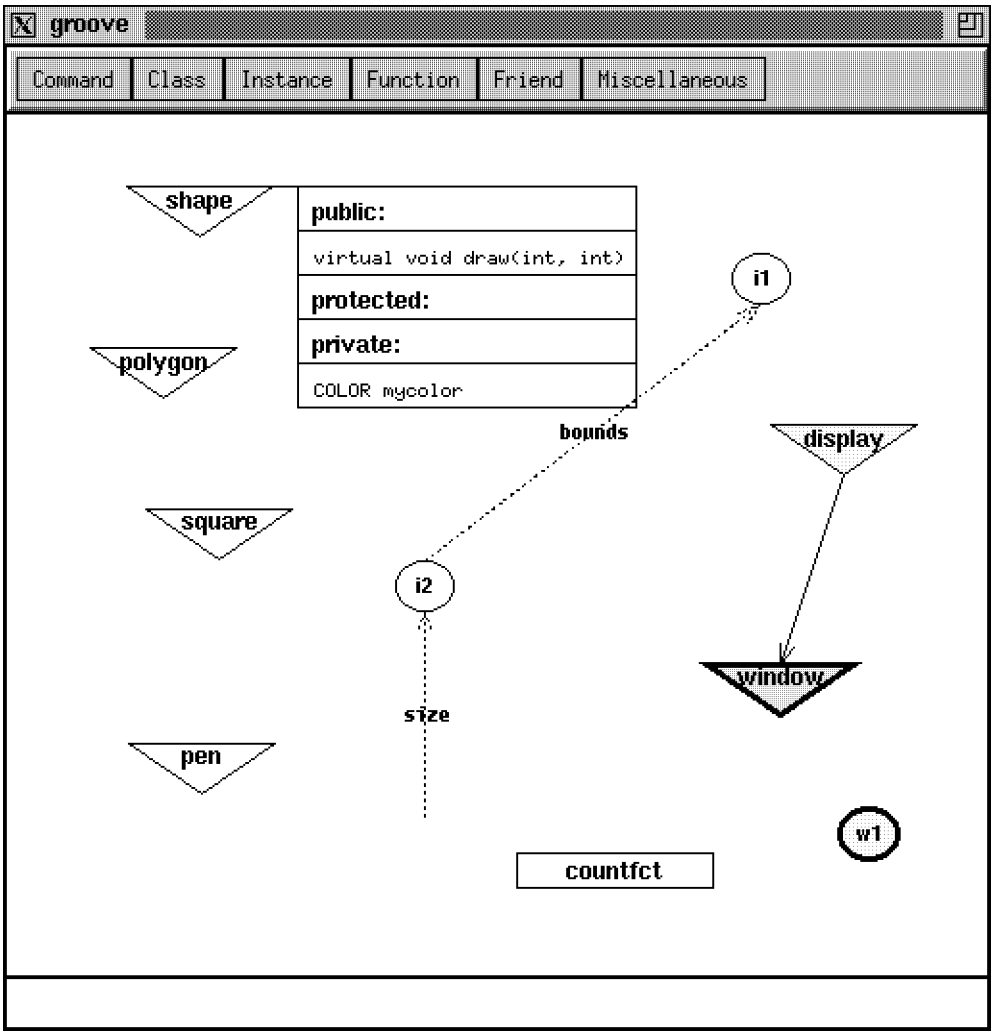

Figure 2: A GROOVE program specification the same as shown in Figure 1 except that focus has been shifted to the "window" class. Note how the other object representations update to illustrate their relationship to the focus entity.

As mentioned above, the amplified visual attributes of an entity are only shown with respect to its relationship to the current focus. For instance, if the current focus is a class, related entities are classes above or below it in the inheritance hierarchy, instances with members received from the class, and friend classes and functions. Table 1 below lists the relationships and visual attributes we currently support for a C++ tool. In Figure 1, focus has been placed on the "polygon" class. In Figure 2, focus is on the "window" class, and in Figure 3 focus is on instance "i2."

These visual attributes are useful for depicting static attributes of programs. To represent the dynamic behavior of programs, however, more is needed. In particular, we utilize animation to reflect run-time behavior and protocols. What exactly constitutes an animation is difficult to define—some would consider a deliberate shuffle among colors for an object to be an animation. We support much more dynamic activities such as objects mov-

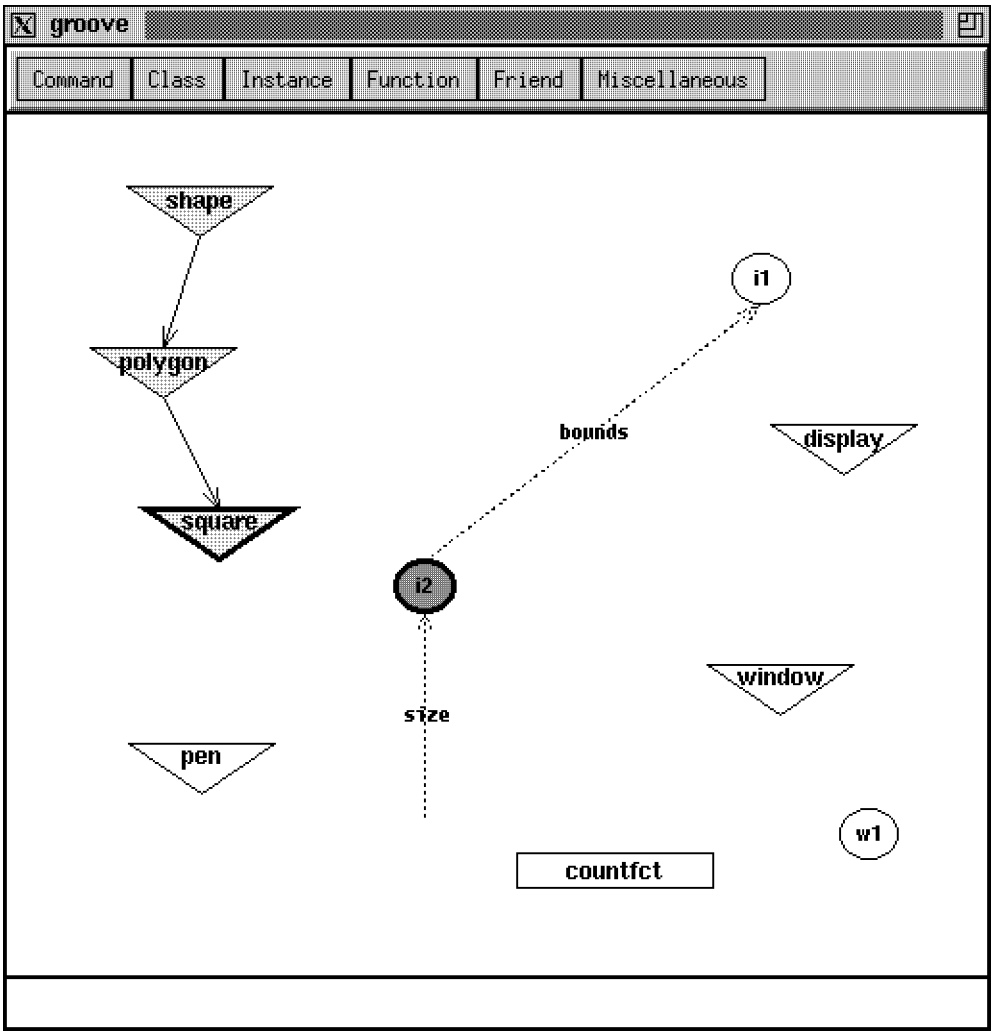

Figure 3: Focus has now been shifted to the instance "i2," and the view of class "shape" has been compressed.

Current focus	Related entity	Entity's depiction (visual attribute)
Class	Itself	Dark color, bold outline
	Base class(es)	Light color, inheritance connection arrows
	Derived class(es)	Light color, inheritance connection arrows
	Instance of the class	Light color, bold outline
	Instance of a derived class	Light color
	Friend class or function	Hand icon
Instance	Itself	Dark color, bold outline
	Its class	Light color, bold outline
	Classes it inherits from	Light color
	Visible instances	Double, broken outline
Function	Itself	Bold outline

Table 1: Graphics representations as relating to a current focus program entity.

ing about a window in a smooth, continuous manner. This provides a true sense of motion, dynamics, and change.

The most direct use of animation in our visual paradigm is in illustrating message traffic between instances. When an instance's method is invoked, we show an arrow smoothly growing out from the current focus to that instance. The arrow is labeled with the name of the message sent. When the method returns, the arrow is slowly "recalled" to its point of origination. If desired, the system can leave a ghost of the arrow as a history of the messages sent. Also planned, but not yet implemented, are the abilities to "carry" other objects such as parameters and return values along with the message, and to indicate from which class a member function invocation is bound. This second capability could be visualized by "lighting up" the classes that an instance derives fields from. Relatively simple notions such as these can provide substantive contributions to specifying and understanding run-time program actions.

We currently use other forms of animation to provide even further information. When object instances are created and constructed, they originally form within a class and smoothly move out of the class to take a position in the program's display. When instances are destroyed, a special "delete" message moves into the display, strikes the instance, and then the instance flares to red and disappears. As the run-time protocol changes and different functions are invoked, instance visibilities dynamically update to reflect the current state.

3 Example Scenario

In this section we walk through two different examples that are intended to motivate the dynamic information provided by our animated approach to design documentation. The examples are based on actual object-oriented systems but are selected for their conciseness in presenting the benefits of animation.

3.1 Dynamic Object Visibility and Lifetimes

The example used in this section is the design of a login protocol for an interactive system. The basic assumption is that there is a workstation in an interactive system on which a user can login and begin work. Access to the workstation is controlled through a *guard* object (an instance of the *Guard* class). The guard object has visibility to a database of valid users which is modeled as a dictionary of user objects, keyed by user id. The initial state of the scenario is illustrated in Figure 4.[1]

The scenario is initiated by sending the *loginUser* message to the guard object. In response to this message the guard object creates an instance of the *LoginWindow* class and sends it the *getUserIdAndPassword* message. The dynamics of these events are

- The receipt of the *loginUser* message and in response:
    1. The creation of a *LoginWindow* object instance,

[1] We utilize diagrams rather than GROOVE window dumps here simply to conserve space.

Figure 5: Login Scenario: Getting Login Information

2. Establishment of visibility for the instance and
3. Transmission of the *getUserIdAndPassword* message to the instance.

Figure 5 shows that the guard object is still in the process of implementing the *loginUser* message (the original message) and the animation has been suspended while the LoginWindow is processing the *getUserIdAndPassword* message.

The return value from the *getUserIdAndPassword* message is two strings representing a user id and a password. The mainline scenario is that the guard object retrieves a user object from the user object dictionary, asks the user object whether the password provided is valid and, if it is, deletes the LoginWindow object and asks the user object to start execution. Statically the Guard *class* has visibility to the User *class* but dynamically the Guard object instance does not have visibility to any User object instance at the beginning of the scenario. Visibility to a single User object instance is established when the User Dictionary returns a User instance as a result of a query. The guard can begin sending messages to the user object once it attains visibility. An important dynamic behavior which is made clear by the animation is that the LoginWindow object is deleted before the *start* message is sent to the user object. Figure 5 shows an animation snapshop as the LoginWindow is being deleted.

There are two possible branches in this animation: The user id may not be valid or the password may not be valid for a particular user. If the user id is not valid then the guard creates an instance of ErrorWindow, sends it the *displayError* message, deletes the ErrorWindow, and re-synchronizes with the protocol by sending the *getUserIdAndPassword* message to the LoginWindow.

If the user Id is valid but the password is not then the guard deletes the user object,

Figure 7: Login Scenario: Invalid User Id

and then displays the error and resynchronizes as above.

Two key ingedients missing in this animation scenario are the depictions of parameters and return values. One possibility for this addition is to show a small (textual) function information window in the corner of the display which is updated appropriately. This solution lacks ties into the existing visualization, however, so that showing an object as a parameter would not be possible.

3.2 Understanding Method-To-Message Binding

An important part of designing the behavior of an individual class is understanding how an instance of the class will interact with itself. A common example of this is in understanding how the components of each class in an object's class hierarchy are initialized. This is a typical *walking up* exercise. Abstract classes can be defined which reference behaviors they expect to be defined in their subclasses (the *Collection* class of Smalltalk-80, for example). In this case the self-reference protocol is *dropping down* the hierarchy. Even without self reference it can often be useful for a developer to understand which class in an object's hierarchy will respond to a particular message. This is particularly important when methods may be selectively redefined and dynamically bound.

As mentioned briefly earlier, we plan to have GROOVE help developers understand message-to-method binding by utilizing the class presentations from which an object inherits fields. When a message is received, the class bindings for receipt of the message could be sequentially "lit up" (not unlike the "Christmas tree" lights for beginning an auto race) to indicate which class' function was actually utilized. This makes clear where in the class hierarchy a message will bind and nicely illustrates self-referential protocols which walk up or drop down the hierarchy.

4 System Design

GROOVE is implemented in C++ on top of the X11 Window System. Each of the two main components, the graphical depiction and its corresponding code, are encapsulated by a class, *GraphView* and *CodeView* respectively. When a user invokes a new command such as **Add Member Function** from the pull-down menu at the top of the window, the user interface portion of the system prompts the user for the appropriate parameters. In this case, the user will pick the class to receive the member function by selecting the class with the mouse. The specifics of the member function, such as name, return type, access, etc., are specified through a dialog box.

From there, the user interface portion of the system invokes the *EventReceive* function of the GraphView and CodeView objects. Currently, each component maintains an abstract representation of the program being built.[2] The GraphView, for example, maintains lists of all visible entities, all classes, all instances, and so on. Each internal object contains fields storing its relationships to other objects. For instance, a class needs to reference its

[2] We are exploring the use of a common abstract model of the program which both components can query in database-style fashion. It is unclear whether the overhead introduced by an extra layer such as this will hinder performance of the GraphView, in which speed is of critical concern.

Figure 8: High-level overview of GROOVE's system configuration.

base classes, derived classes, instances, friends, etc. An overview of GROOVE's system configuration is shown in Figure 8.

We implement the graphics and animation portion of GROOVE using a derivation of the path-transition animation paradigm[Sta90]. The paradigm allows us to query image positions and construct paths between any pair of these positions. Finally, movement actions, images sliding along the pertinent paths, allow us to create the animations.

5 Run-time Visualization

5.1 Animation of Existing Systems

The animation techniques used in GROOVE can be used not only for designing protocols but also for viewing the run-time behavior of an existing system. This can be useful for understanding the behavior of an undocumented system or for understanding whether a system follows expected messaging patterns.

In order to animate an existing system we must augment it with GROOVE animation machinery. This is currently done by processing the C++ source files and augmenting each member function with additional source.[3] The most important thing to know for the

[3]This parsing capability is extremely difficult without the equivalent of a full compiler. Currently, our

purposes of animation is when member functions are entered and exited. We accomplish this by introducing an additional object instance into each member function as the first declaration in the source of the function implementation. C++ semantics ensure that this object instance is created when the function is entered and is destroyed when the function exits. The sole purpose of the object instance is to record member function entry and exit. The constructor for the object generates a member function entry event which the animation system displays as a message send from the current object focus to this object. The current object focus becomes the object receiving the message. We know that the method has exited when the destructor is called for the tracing object that was introduced. The destructor for that object informs the animation system that the member function has exited. The system allows new objects to either be placed manually or to be automatically positioned. Currently, the automatic layout algorithm is quite primitive, and it is one area ripe for future improvement.

In the code that follows we show a source file after it has been augmented for animation. The animation script adds animation headers at the top of the source and _TraceObj objects (instance of the TraceMethod class) in each method.

```
#include "animatelib.h"
#include "groove.H"
#include "grooveview.H"
#include "trace.H"
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include "String.H"

String::String(void)
{
TraceMethod _TraceObj("String", "String(void)", this);
    s = NULL;
}

String::~String(void)
{
TraceMethod _TraceObj("String", "~String(void)", this);
    if (s != NULL)
        delete s;
}

String &
String::operator=(String &aString)
{
TraceMethod _TraceObj("String", "operator=(String &aString)", this);
    if (&aString == this)
        return *this;
    else
        return operator=(s);
}
...
```

Even with this relatively simple example we can see how animation can help in program debugging. Consider the member function **String::operator=(String &aString)**.

techniques are rather limited–we use AWK to parse the source files.

The expected behavior of this member function is that it will send a message to itself to implement the assignment operator. If the argument is also the receiver, however, then the member function simply returns a reference to itself without sending the additional message. This difference in behavior would be difficult to notice in a debugger trace but the different visual pattern will be immediately noticeable.

6 Further Related Work

Cunningham and Beck created a system for diagraming object-oriented computations, primarily displaying message passing and inheritance among Smalltalk classes[CB86]. They used a box (class) and arc (message) notation in order to help teach object-oriented programming. Although they briefly experimented with adding dynamic behavior, their system displayed static imagery without timing or sequencing information.

Rumbaugh uses state diagrams (also called statecharts[Har87]) as part of dynamic models for describing the states that an object instance can enter and how the states change in response to events. The objectcharts of Coleman, Hayes, and Bear [CHB92] extend the notion of statecharts to include the effect of transitions on attributes and to take into consideration messages and subtyping. Both sets of authors suggest that the aggregate state of a system be viewed as the union of the statecharts (or objectcharts) of the object instances in the system. We view objectcharts as a useful design and specification technique that is at a lower detail level than we currently support. We also feel the techniques will be enhanced by the ability to animate the state changes that can occur within the operation of a system.

Kleyn and Gingrich sought to go beyond static displays by examining the dynamic behavior of object-oriented systems written in a Common Lisp-style language[KG88]. Their GraphTrace tool illustrated structural and behavioral views of object-oriented systems by recording message traffic for subsequent replay. The tool's displays mainly involved graph diagrams consisting of nodes and arcs. Animation, however, was restricted to simply highlighting and annotating graph nodes. Böcker and Herczeg provide more extensive animation of Smalltalk-80 traces with the Track system[BH90]. Track allows programmers to visually specify message tracing as a debugging aid. At execution time, the system presents an animation of the messages sent between objects. These systems give animation of existing programs, but they fail to give an emphasis to the proactive use of animation in the design of object-oriented systems.

The commercial system ObjectCraft[HS91] supports graphical design of object-oriented programs with post-design code generation. GROOVE differs from ObjectCraft via our "on-the-fly," rather than post-design, code generation and our inclusion of animation to specify program dynamics.

7 Summary

Various groups have taken different approaches to diagramming and documenting dynamic object-oriented system behavior, but their efforts have been limited by the underlying technology on which they are based. Advances in program animation techniques have allowed

us to construct a tool which facilitates designers describing the dynamics of their systems in a medium which naturally reflects the manner in which they are conceived. With such a tool we move past the paradigm of *what-you-see-is-what-you-get* to *what-you-want-to-see-is-what-you-get-to-see*. The use of animation in documentation also provides high information density allowing systems dynamics to be described succinctly with high comprehension. We have embodied our animated design and documentation techniques in an object-oriented system development tool called GROOVE.

One final note: The description of GROOVE in this paper suffers from the very problem that the system is designed to address! We are using a static medium (paper and text) to describe a dynamic, time-changing entity. To truly appreciate the capabilities GROOVE offers, the system must be seen live.

References

[BC89] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings of the ACM OOPSLA '89 Conference*, pages 1–6, New Orleans, LA, October 1989.

[BH90] Heinz-Dieter Bocker and Jurgen Herczeg. What tracers are made of. In *Proceedings of the ECOOP/OOPSLA '90 Conference*, pages 89–99, Ottawa, Ontario, October 1990.

[Boo91] Grady Booch. *Object Oriented Design with Applications*. Benjamin Cummings, 1991.

[CB86] Ward Cunningham and Kent Beck. A diagram for object-oriented programs. In *Proceedings of the ACM OOPSLA '86 Conference*, pages 361–367, Portland, OR, September 1986.

[CHB92] Derek Coleman, Fiona Hayes, and Stephen Bear. Introducing objectcharts or how to use statecharts in object-oriented design. *IEEE Transactions on Software Engineering*, 18(1):9–18, January 1992.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HS91] Paul Harmon and Brian Sawyer. *ObjectCraft: A Graphical Programming Tool for Object-Oriented Applications*. Addison-Wesley, Reading, MA, 1991.

[KG88] Michael F. Kleyn and Paul C. Gingrich. GraphTrace - understanding object-oriented systems using concurrently animated views. In *Proceedings of the ACM OOPSLA '88 Conference*, pages 191–205, San Diego, CA, September 1988.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, New York, NY, 1991.

[Sta90] John T. Stasko. The Path-Transition Paradigm: A practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3):213–236, September 1990.