Visualizing the Behavior of Object-Oriented Systems

Wim De Pauw, Richard Helm, Doug Kimelman, and John Vlissides
IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598 USA
{wim,helm,dnk,vlis}@watson.ibm.com

Abstract

Numerous classes, complex inheritance and containment hierarchies, and diverse patterns of dynamic interaction all contribute to difficulties in understanding, reusing, debugging, and tuning large object-oriented systems. To help overcome these difficulties, we introduce novel views of the behavior of object-oriented systems and an architecture for creating and animating these views. We describe platform-independent techniques for instrumenting object-oriented programs, a language-independent protocol for monitoring their execution, and a structure for decoupling the execution of a subject program from its visualization. Case studies involving tuning and debugging of real systems are presented to demonstrate the benefits of visualization. We believe that visualization will prove to be a valuable tool for object-oriented software development.

1   Introduction

Understanding the structure and internal relationships of large class libraries, frameworks, or applications is essential for fulfilling the promise of code reuse. Moreover, discerning global and local patterns of interaction among classes is critical for tuning and debugging. Although the object-oriented paradigm lets programmers work at higher levels of abstraction than procedural models, the tasks of understanding, debugging, and tuning large systems remain difficult. This has numerous causes: the dichotomy between the code structure as hierarchies of classes and the execution structure as networks of objects; the atomization of functionality—small chunks of functionality dispersed across multiple classes; and the sheer numbers of classes and complexity of relationships in applications and frameworks.

Tools for procedural languages are often inappropriate for object-oriented programs because they work at

an inappropriate level of abstraction. Tools that *are* directed at object-oriented software development have focused primarily on static code structure (for example, class browsers and inheritance viewers [10, 4, 15], affinity browsers [7]) and on breakpoint debugging and object inspection [1], which provide only microscopic views of the program at disjoint points in time.

We believe tools that focus on the dynamic behavior of an object-oriented system are essential for understanding, code reuse, debugging, and tuning. We also believe that visual tools are most effective for this purpose. Users are easily overwhelmed by a steady stream of text. The fields of scientific visualization and program visualization have demonstrated repeatedly that the most effective way to present large volumes of data to users is in a continuous visual fashion [17, 14, 11]. Continuous visual displays allow users to assimilate information rapidly and to readily identify trends and anomalies.

This paper introduces a system for dynamic visual presentation of the behavior of object-oriented systems. Major contributions of this work are a set of novel views for displaying system behavior and a flexible distributed architecture for animating the views based on program executions. The set of views includes displays that cluster classes based on the degree to which they interact, histogram variants showing class instances and their activity levels, and cross-reference matrices indicating the degree of various forms of inter- and intra-class references.

Important goals of the system's architecture include:

1. *Versatility.* Allow users to observe and visually inspect a system, either in real-time or post-mortem, from both a local and global perspective.

2. *Composability.* Allow users to combine different instrumentation, analyses, and views, all to be active concurrently.

3. *Extensibility.* Allow users to add new visualizations to the system quickly and easily. The set of visualizations must not be fixed. This implies a toolkit approach to building visualizations.

4. *Platform and language independence.* The instrumentation technique and protocol for describing program behavior should be language- and platform-independent. Although this paper focuses on the visualization of programs written in C++, the choice of language is not intrinsic to this architecture.

With this system, applications as well as frameworks or class libraries are first instrumented by a preprocessor. As an instrumented application executes, it produces an event stream that characterizes its progress. A visualization system reads this event stream and updates its model of relevant aspects of the executing program. This model then drives a number of user-selectable views of program behavior.

This paper describes the visualization architecture in general, key aspects of our prototype implementation, and some views we have developed. The next section presents actual use of the system on a real application to give a feel for the system's effectiveness. Section 3 describes the overall visualization architecture and some of its unique features. Section 4 presents techniques used to instrument code and to record events. Section 5 introduces the event language Annotalk for communicating between elements in this architecture. Sections 6 and 7 discuss issues in the design of the models that drive the views and the Visualizer classes that actually produce the views. We conclude with comparisons to related work and future directions for this work. An appendix describes some predefined views currently available with the system.

2 A Visualization Case Study

During the development process, a programmer turns to visualization either for general inspection to verify that a program is running smoothly or to track down the cause of a problem that has arisen. Typically, high-level views are examined first to get an overview of system behavior, and then more specific views are used to focus on suspicious behavior.

This section presents actual experience with visualization in the development of QOCA, a constraint-solving toolkit [6]. We begin by examining a high-level view that shows overall patterns of communication.

2.1 Visualizing Communication

The **inter-class call cluster** provides a dynamic overview of communication patterns between classes. Figure 1.1[1] shows a snapshot of this view early in the

---

[1]Color plates of the visualizations in this paper appear at the end of the proceedings.

execution of QOCA. This view shows class names as floating labels. The amount of communication between instances of two classes determines the distance between their labels. The view is animated so that the more communication there is between classes, the more their labels gravitate towards each other and cluster together. Classes that communicate infrequently are repelled towards the edge of the view.

For QOCA, the classes Term, Terms, TermsIterator, Factor, Factors, and FactorsIterator, concerned with representing and manipulating constraints, clearly show strong interaction with each other.

This view also indicates the current call stack by showing the classes of instances that have received messages on the call stack. A blue path leads from the label ::main through each of these classes. The last segment of the path, leading to the currently active class, is red. In Figure 1.1 the thread of control goes from ::main, through Objective, CompoundExpRep, and Terms, and finally to the currently active class TermsIterator.

The inter-class call cluster focuses attention on the most active and most cooperative classes at any moment. These classes provide a good starting point for more detailed study either for optimization or understanding the structure of an application—clustered classes, for example, are likely to be tightly coupled or from the same subsystem [21]. The number of classes in a cluster is typically small, on the order of ten classes or fewer, probably because systems with broader interactions are exponentially more complex and are less likely to be developed in the first place.

Object-oriented programs often exhibit distinct execution phases. Most programs have at least one initialization phase as a precursor to a (much longer) communication phase; programs may have several such phases. Different phases become evident from the dynamics of the inter-class call cluster. A new phase starts when many new classes burst out of the center of the view. Some classes gravitate together quickly; other migrate to the edges of the view. The call stack path also reflects a new phase when its shape changes drastically after a period of relative stability.

An execution hot-spot often manifests itself when the red (active) portion of the call stack path darts between the same set of classes for long periods. Such classes are prime candidates for optimization. Paying particular attention to small but popular classes can be more effective than redesigning complex but infrequently used classes. In this example, the classes Term, Terms, TermsIterator, Factor, Factors, and FactorsIterator are good prospects for performance tuning.

327

2.2 A Closer Look at Communication

While the inter-class call cluster offers insight into the dynamic messaging behavior of the program, the inter-class call matrix (Figure 1.2) gives cumulative and more quantitative information. Classes appear on the axes in the order in which they are instantiated. Base classes always appear closer to the origin than their subclasses. A colored square in this visualization represents the number of calls from a class on the vertical axis to a class on the horizontal axis. The color key along the bottom indicates relative number of calls. Colors range from red, denoting fewer calls, to violet and ultimately black, denoting more calls.

The dark squares in this view confirm our impression from the inter-class call cluster that Term, Terms, TermsIterator, Factor, Factors, and FactorsIterator are called most frequently. As mentioned, it often pays to take a closer look at classes showing high activity. Such classes are often the key to understanding larger parts of the system and to optimizing its performance. Moreover, unexpectedly high activity can be symptomatic of bugs in the program.

Many inter-class dependencies can appear as macroscopic features in the inter-class call matrix. Vertical stripes indicate classes that are called by many other classes. Vertical stripes appearing above the diagonal tend to indicate key abstract classes in the framework or library. Horizontal stripes indicate a class that calls many other classes, typically the classes of its instance variables. Clusters close to the diagonal may indicate tightly coupled classes or subsystems. The appendix discusses this visualization in more detail.

2.3 Insight from Instances

The two previous visualizations primarily concern displaying relationships between classes. Focusing on instances shows program structure at finer levels of granularity.

The **histogram of instances** (Figure 1.3) displays all instances of each class. Rows of small colored squares form the bars of the histogram. Each bar represents all instances of the class whose label appears to its left. Again, a square's color indicates the number of messages an instance has received. Colored squares appear and disappear as objects are instantiated and destroyed. White squares indicate objects that have been destroyed; these squares will be reused by newly created instances. This visualization lets us see how many instances exist at a given time and their level of messaging activity. It also shows relative object lifetimes and anomalies such as undesired copy constructor calls that are manifest as extremely short-lived objects.

Consider again the classes Term, Terms, TermsIterator, Factor, Factors, and FactorsIterator. This view shows clearly that Term, Factor, and Factors have unexpectedly large numbers of instances, possibly indicating a memory leak. Indeed, waiting until the application terminates verifies that most instances of these classes are never reclaimed.

2.4 Pinpointing the Problem

To correct this memory leak, a first step might be to find the classes responsible for allocating these unreclaimed instances. The **allocation matrix** (Figure 1.4) plots classes that allocate new objects versus the classes they instantiate. This view shows allocation dependencies and the most frequently allocated objects. We can use this information both to pinpoint the sources of allocations and to subsequently reduce storage and construction costs.

In this case it appears that Term is allocating most instances of Term and Factors. This is good evidence that the class Term fails to reclaim these instances.

Glancing back at the inter-class call matrix (Figure 1.2), we see that two classes send messages to Term most frequently: Term itself and Terms. Inspecting the code we find that Term passes the unreclaimed instances to Terms, which never deletes them. Correcting this bug by adding a missing "delete" statement leads to the healthier histogram of instances in Figure 1.5, shown at the same point in the program's execution.

3 Architectural Overview

Program visualization involves instrumenting a **subject program** so that it generates **events** of interest during execution. As the subject runs, a **visualization application** interprets these events and builds **models** of the subject's state. The visualization application uses these models to drive **visualizers** that present **views** reflecting the subject's behavior.

Our visualization architecture provides versatile, platform-independent, composable, and extensible visualizations of object-oriented systems by partitioning functionality into four components:

1. **Instrumentation** augments the subject program with code that generates events. It also adds an **instrumentation run-time** that transmits events to the visualization application and lets it access and explore the subject's internal state and control the subject's execution.

2. **Communication** defines a language-independent protocol and transport mechanism between the subject and its visualization application.
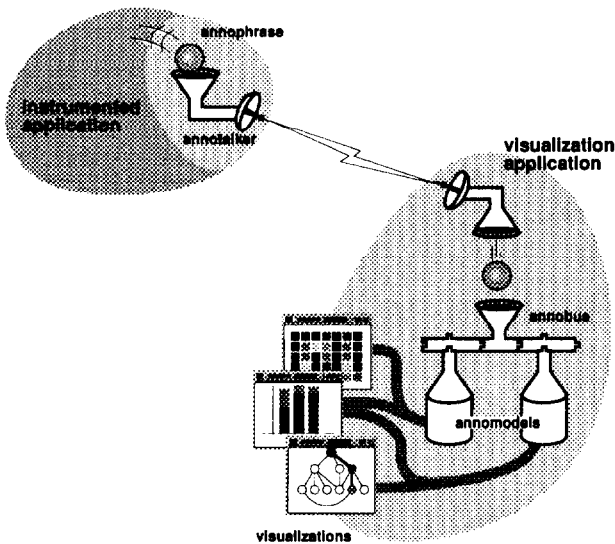
Figure 1: Architectural overview

Figure 2: Instrumentation process

3. **Modeling** assimilates events into models that represent and track the execution behavior of the subject. Models make it convenient for visualizers to access and display this information.

4. **Presentation** constructs views of program behavior that let users explore the subject's behavior and internal structure.

Figure 1 provides an overview of the objects that implement the architecture. We will describe and refer to these objects in the detailed discussions that follow.

4 Instrumentation

The subject's instrumentation must support effective visualization. For object-oriented programs this requires that the instrumented code

- generates events for object construction and destruction;

- generates events for method entry and exit;

- collects static type information, such as class structure and member declarations;

- collects dynamic type information to resolve an instance's class at run-time;

- supports suspending and resuming subject execution;

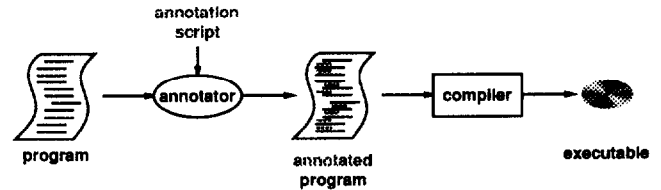- lets visualization applications examine and explore a subject's internal state.

There are many ways to instrument the subject to fulfill these requirements. Our implementation annotates C++ source code with instrumentation code. We will discuss the benefits and liabilities of the annotation approach later in this section.

Figure 2 shows the steps in the instrumentation process. A script-driven annotator analyzes the subject source code and augments it with instrumentation code. The result is then compiled and linked to form a new binary, which is executed normally.

The instrumentation collects static type information before executing subject code; dynamic type information is generated as the subject runs. The communication component of the architecture transmits this information to visualization applications, as we describe in Section 5.

4.1 Run-time Type Information

To generate meaningful events, the subject program must track member function entry/exit and constructor/destructor calls. It must also maintain a run-time type system. C++ does not provide type information at run-time (though proposals for such a mechanism exist [16]); therefore the instrumentation run-time must support this functionality. Our implementation is based on Interrante and Linton's work in this area [9].

For each class in the subject program the instrumentation maintains an object that records static and dynamic type information about the class. Each of these objects is an instance of a **Blues**[2] subclass. Blues is an abstract base class with protocol for identifying a class's parents, members, and instances. We define one Blues subclass for each class in the subject program. There exists one instance of each Blues subclass in the instrumented code to maintain this information.

Figure 3 shows the Blues interface and a subclass **ABlues** for a class named **A** in the subject program. Given an arbitrary instance from the subject program, the Blues base class returns the corresponding Blues subclass instance that can furnish type information at run-time. For example, given a pointer to an instance of class **A**, the function **Blues::Instance(A*)** returns

---

[2]What's in a name?

```
class Blues {                                     class ABlues : public Blues {
public:                                           public:
    virtual ~Blues();                                 static ABlues* Instance();

    static Blues* Instance(void*);                    virtual void* Member(void*, const char*);
    static Blues* Instance(const char*);
                                                      virtual const char* Class();
    void Register(void*, const char*);                virtual const char** Parents();
    void Unregister(void*, const char*);              virtual const char** Members();
    void Register(void*);                         protected:
    void Unregister(void*);                           ABlues();
                                                  private:
    void Entering(void*, const BluesInfo&, boolean);      static ABlues* _A_Blues;
    void Leaving(void*, const BluesInfo&, boolean);       static const char* _AMembers[];
                                                      static const char* _AParents[];
    virtual const char* Class() = 0;                  static const char* _AClassName;
    virtual const char** Parents() = 0;           };
    virtual const char** Members() = 0;
    virtual void* Member(void*, const char*) = 0;
protected:
    Blues(const char *);
    // ...
};
```

Figure 3: Blues base class (left) and annotator-generated subclass interfaces (right)

a pointer to the ABlues object that records information about the A class. The ABlues instance may then be queried for its class name (via the Class operation), its parents (via Parents), and its member signatures (via Members). Moreover, the Member operation lets a client ask the ABlues for a pointer to a member of this instance given the member's signature. This operation lets visualization applications examine an object's contents and follow pointers to other objects, even private ones.

To make the private member data of A accessible to ABlues, we must circumvent C++'s type system. Each Blues subclass uses a slightly modified declaration of its associated subject class, one that declares the Blues subclass to be a friend of the subject class. The ABlues implementation, for example, sees a version of the declaration of class A with an additional "friend class ABlues;" statement, which lets ABlues access member data of an instance of A. Of course, this technique assumes that friend declarations do not alter class layout, which appears true for all compilers we know.

The Blues base class and subclasses are part of the instrumentation run-time. They are compiled and linked separately from the annotated subject code. Only the annotations in the subject code use Blues services, which are transparent to subject code.

Unlike some proposals for maintaining run-time type information, the subject classes are not queried for their type directly in this scheme. Instead, Blues's static Instance operations return the particular Blues subclass object given a specific instance or a class name. An advantage of this approach is that it is not invasive—it depends only on class declarations, not implementations. It does not alter the memory layout or virtual function

tables of subject classes. This makes it possible to query classes defined in a library or toolkit about their structure even if the library does not provide source code.

4.2 Construction, Destruction, and Method Invocations

To generate events during subject execution, we add code to each class's constructor, destructor, and member functions. We track object construction and destruction by adding code that registers and unregisters each instance with the appropriate Blues subclass instance (see Figure 3). Events are generated as a side-effect of (un)registering an instance with the Blues subclass.

We track function entry and exit by declaring a local (stack-allocated) instance of **BluesTracer** at the beginning of each constructor, destructor, and member function. BluesTracer is a trivial class whose constructor and destructor inform the corresponding Blues subclass instance of the member function's entry and exit. This generates an event. Because the BluesTracer instance is stack-allocated, its constructor and destructor will be called automatically when the member function enters and exits.

4.3 Suspending and Resuming Execution

Objects in the instrumentation run-time are responsible for suspending and resuming execution of the subject program. In the current implementation, control over execution is carried out entirely as a side-effect of communication, as we describe in Section 5.

4.4 Discussion

Instrumentation by annotation has several advantages. It is independent of operating system, compiler, linker, and execution format, thus enhancing portability. Because annotation is automatic, its effects are transparent to the programmer; the subject program need not be modified. Annotation also offers flexibility. Annotations can be changed simply by changing the script, and the program can furnish any information that it can compute at run-time.

A potential disadvantage of this scheme is that instrumentation code may have undesirable side-effects on the subject, such as slowing execution or increasing storage requirements. However, the annotation approach is not necessarily more disruptive than environment-specific schemes, since all perturb the subject to some extent.

One shortcoming of our approach is its inability to track data member references. This is a consequence of our simple-minded annotation script, not the annotation approach. A smarter parsing script coupled with additional instrumentation code would yield data reference information, presumably with significant run-time overhead.

5 Communication

The architecture is designed so that subject programs are independent of visualization applications. The subject and the visualization application may run in different address spaces or on different machines. The instrumented subject includes communication objects that send information to a visualization application and receive requests from it. The communication component of the architecture consists of (1) **Annophrase** objects, which encapsulate program events, static program information, and control directives; (2) **Annotalker** objects, which logically send and receive annophrases; and (3) **Annotalk**, a protocol for transmitting annophrases between annotalkers.

Figure 1 shows an annotalker in the subject instrumentation run-time sending an annophrase to a visualization application's annotalker. Visualization applications also use annotalkers to communicate with their subject, for example, to send a control directive or to request run-time type information. From the instrumentation's perspective, annotalkers send and receive annophrases, but annotalkers use Annotalk to encode and transmit annophrase information.

Annotalk is a simple, two-way protocol that can convey information in an annophrase. Table 1 lists statements in Annotalk and describes them briefly. Most are self-explanatory. The rh, rr, and rm statements encode requests for stopping and restarting execution and

for returning a pointer to a member datum given the datum's signature (including its name and type declaration). The x statement defines a dictionary entry that maps an arbitrary signature into an index. Long signature arguments are encoded as much shorter dictionary indices, thereby compressing the protocol stream for efficient transmission and storage.

With Annotalk, annotalkers can transmit annophrases across process and machine boundaries, either through files or interprocess communication. They also perform transparent compression and decompression of the Annotalk stream as directed by dictionary entry declarations and accesses. Finally, an annotalker blocks the subject program when it receives a halt request, responding only when it receives a request for a pointer to a member or a request to resume execution; other annophrases are ignored.

6 Modeling

Visualization applications must maintain knowledge about the subject program's execution. This knowledge is stored in one or more **annomodels**. Annomodels assimilate and distill execution events from the subject into visualization-specific information. Annomodels receive events in the form of annophrases from the subject as it executes. In turn, annomodels drive visualizers that present views.

Annomodels may be as general or as specialized as a visualization warrants. Subclasses of the Annomodel base class process and maintain different kinds of information. For example, the ClassModel subclass stores static information about classes, including a catalog of all classes, their inheritance relationships, and member information. InstanceModel maintains lists of all instances, the instances of each class, and a count of messages to each instance. Other annomodels can collect call stack information, keep track of logical time, provide an interface for controlling program execution, and compile annophrase statistics such as the amount of protocol compression being realized. Annomodels can also synthesize higher-level, visualization-specific events from multiple annophrases.

A visualizer may rely upon several annomodels for information, and there may be multiple synchronized visualizers. Concurrent views provide several perspectives on a program's execution and give insight greater than what views offer in isolation. Each visualizer registers itself with the annomodels it needs; annomodels in turn update all registered visualizers. Moreover, visualizers often share annomodels because they require similar sorts of information to generate views. For ex-

Annotalk statement	Description
cl *class_name*	declares a class name
pubp *class_name parent_class_name* prop *class_name parent_class_name* prip *class_name parent_class_name*	public derivation protected derivation private derivation
pubf *class_name signature* prof *class_name signature* prif *class_name signature*	public member function of *class_name* protected member function of *class_name* private member function of *class_name*
pubd *class_name signature* prod *class_name signature* prid *class_name signature*	public data member of *class_name* protected data member of *class_name* private data member of *class_name*
c *time_stamp object_ptr class_name source_file source_line signature* d *time_stamp object_ptr class_name source_file source_line signature* e *time_stamp object_ptr class_name source_file source_line signature* l *time_stamp object_ptr class_name source_file source_line signature*	constructor call destructor call member function entry member function exit
rh rr rm *object_ptr signature* m *object_ptr signature*	request to halt execution request to resume execution request for pointer to member *signature* of *object_ptr* response to pointer request
v *version_number* x *code string*	declares the current protocol version declares a dictionary entry, accessed by @*code*

Table 1: Annotalk protocol

ample, most visualizers use a ClassModel and an InstanceModel.

Each annomodel is attached to an **annobus**, which notifies the annomodels of an annophrase's arrival and gives each a chance to examine the annophrase and update its state. Of course, annomodels are free to ignore annophrases. Annomodels are notified once again when the annophrase has been inspected by all annomodels. Only then do annomodels update their attached visualizers. This two-phase notification serves two purposes: it lets annomodels update their respective visualizers only when they are consistent with one another, and it gives cooperating annomodels a chance to examine the annophrase before they exchange information.

The more focused an Annomodel subclass is, the greater its potential for reuse and combination with other annomodels. While visualization-specific Annomodel subclasses can be many and varied, there should be little or no overlap in the processing or storage responsibilities of any two annomodels. Redundancy can lead to inconsistencies (e.g., two annomodels disagree), inefficiency (e.g., replicated effort), or both.

7 Presentation

The visualization architecture separates modeling data kept in annomodels from presentations defined by visualizers. All visualizers are derived from the Visualizer base class. Visualizer defines protocol that lets an annomodel notify it of state changes. It also defines protocol for interpreting basic program events such as constructor and destructor calls and method entry/exit.
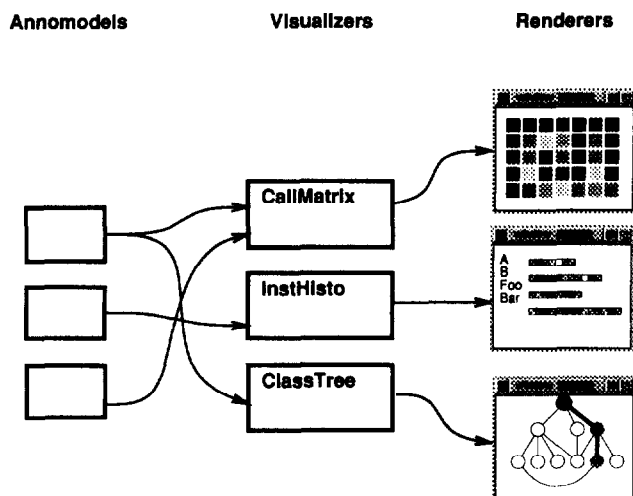
Annomodels   **Visualizers**   **Renderers**

Figure 4: Annomodel-Visualization-Renderer relationships

Subclasses of Visualizer refine the interpretation of program events and generate specialized views.

Each visualizer also maintains its own data to support its visualization. This data will typically map information kept in annomodels to information presented on the display. For the graphical presentation of the view, each visualizer maintains a **Renderer** class (see Figure 4). Renderer defines a high-level interface for displaying and changing graphical elements, thereby isolating visualizers from details of the underlying graphics system. Our prototype implementation uses the InterViews and Unidraw toolkits and ibuild interface builder to implement renderers [13, 18, 19].

We have developed several abstract subclasses of Visualizer. The matrix views in the Appendix are all simple extensions of **MatrixVisualizer**, a subclass for scatter-plot visualizations. **TimeChartVisualizer** is a subclass for strip-chart visualizations that map time to a spatial dimension. **ClusterVisualizer** displays entities clustered according to subclass-defined criteria. Finally, **HistogramVisualizer** uses a histogram to organize information spatially. Programmers can select one of these abstract base classes based on the look and feel they desire. Then the programmer must simply redefine how the visualizer maps program semantics stored in annomodels into the graphical elements of the Visualizer subclass.

A key principle in designing visualizations of object-oriented programs in particular is to take advantage of the structure this paradigm affords. Visualizations can map class/method/instance/time dimensions to spatial/temporal/color dimensions in any combination to produce different views, some more effective than others. The challenge is to choose the most effective mapping for conveying a given aspect of a program's behavior.

8   Related Work

The notion of program visualization first appeared in the literature over a decade ago [8], and algorithm animation was popularized shortly thereafter [2]. Work on visualizing the dynamics of program execution has flourished in the area of parallel systems [11, 5], where the need is clear for a means of understanding the interactions among elements of a complex system.

More recently, the importance of program visualization for object-oriented systems has been recognized.

For purposes of debugging, Bocker and Herczeg [1] introduce a "software oscilloscope" for visually tracking the detailed interactions between objects in a system. To inspect dynamic program behavior of a program during execution, they introduce obstacles between objects and animate the flow of messages across these obstacles. At any point, execution can be suspended to inspect context with conventional browsers. However, focus is solely on debugging and microscopic program behavior. No consideration is given to higher-level program structure or more global behavior over time.

Wilde and Huitt [20] suggest that a major barrier to maintaining object-oriented software is difficulty in program analysis and understanding, and they recommend that visual tools be developed to aid in these activities. In particular, the recommendations include tools based on dependency analysis, graphs, and clustering methodologies, which might help address the problems of comprehending high-level system structure, dynamic binding, and dispersed program structure, among others.

Davis and Morgan [3] introduce one formulation of a graph showing invocations between methods. Their primary interest is in revealing behavior reuse. Reuse of behaviors manifests itself as imbalance in the graph, with a preponderance of edges leading down and to the left. Low-level or basic behaviors drift to the bottom of the graph, and high-level or application behaviors rise to the top. While this view provides an excellent indication of reuse, it would be difficult to gain much understanding of overall system behavior or class-level interaction from its use in isolation.

Kleyn and Gingrich [12] address the need for understanding object behavior in order to facilitate code sharing and reusability. A tool is presented for concurrently animating a number of different graph-based views of the dynamic behavior of an object-oriented program. Structural views include excerpts of the inheritance and containment hierarchies. Behavioral views include graphs of invocations between methods as well as invocations between objects and invocations between methods associated with objects. However, animation is achieved solely by highlighting nodes on fixed graphs. The graphs themselves are not dynamic in that their form never evolves. Further, no consideration is given to presenting other aspects of run-time behavior, such as object allocation activity and object lifetime.

9   Conclusion

We have introduced novel views of the behavior of object-oriented systems and an architecture for creating and animating these views. We have shown how instrumentation can be done in a portable fashion and how a visualization system can be structured so that new visualizations can be added quickly and easily. Our visualizations have already proved effective in our day-to-day work for understanding large, complex object-oriented systems and for debugging and tuning them.

We are continuing to define a comprehensive set of views dealing with a broad range of aspects of object-oriented system behavior. We are also expanding the set of object-oriented languages with which this system can be used. Moreover, we are investigating architectural support for navigation between views, exploration and elision of details within views, and new tools that will enable users to explore the internal state and structure of executing applications. We believe that visualization of object-oriented systems will be an effective and complementary addition to existing development tools. The

architecture and views we have presented lay a foundation for these visualizations.

References

[1] H.D. Bocker and J. Herczeg. Browsing through program execution. In *INTERACT '90*, pages 991–996. Elsevier Science Publishers B.V. (North Holland), 1990.

[2] M.H. Brown and R. Sedgewick. A system for algorithm animation. In *ACM SIGGRAPH '84 Conference Proceedings*, pages 177–186, 1984.

[3] J. Davis and T. Morgan. Object-oriented development at Brooklyn Union Gas. *IEEE Software*, 10(1):67–74, 1993.

[4] Adele J. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, MA, 1984.

[5] M.T. Heath and J.A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):23–39, 1991.

[6] Richard Helm, Tien Huynh, Kim Marriott, and John Vlissides. An object-oriented architecture for constraint-based graphical editing. In *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics*, pages 1–22, Champéry, Switzerland, October 1992. Also available as IBM Research Division Technical Report RC 18524 (79392).

[7] Richard Helm and Yoëlle S. Maarek. Integrating information retrieval and domain specific approaches for browsing and retrieval in object-oriented class libraries. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 145–161, Phoenix, Arizona, October 1991. ACM.

[8] C.F. Herot, G.P. Brown, R.T. Carling, M. Friedell, D. Kramlich, and R.M. Baecker. An integrated environment for program visualization. In H.-J. Schneider and A. J. Wasserman, editors, *Automated Tools for Information Systems Design*, pages 237–259. North Holland Publishing Company, 1982.

[9] John A. Interrante and Mark A. Linton. Runtime access to type information in C++. In *Proceedings of the 1990 USENIX C++ Conference*, pages 233–240, April 1990.

[10] Sharam Javey, Kin'ichi Mitsui, Hiroaki Nakamura, Kazu Yasuda Tsuyoshi Ohira, Kazushi Kuse, Tsutomu Kamimura, and Richard Helm. Architecture of the XL C++ browser. In *Proceedings of CASCON '92*, Center for Advanced Studies. IBM Canada. Toronto. Canada, November 1992.

[11] D.N. Kimelman and T.A. Ngo. The RP3 program visualization environment. *The IBM Journal of Research and Development*, 35(6), November 1991.

[12] M.F. Kleyn and P.C. Gingrich. Graphtrace—understanding object-oriented systems using concurrently animated views. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 191–205, 1988.

[13] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.

[14] G.M. Nielson, B.D. Shriver, and J. Rosenblum. *Visualization in Scientific Computing*. IEEE Computer Society Press, Washington, 1990.

[15] Harold L. Ossher. Multi-dimensional organization and browsing of object-oriented systems. In *Proceedings of the IEEE Computer Society 1990 International Conference on Computer Languages*, pages 128–135, New Orleans, LA, March 1990.

[16] Bjarne Stroustrup and Dmitry Lenkov. Runtime type identification for C++. *Journal of Object-Oriented Programming*, pages 32–42, March-April 1992.

[17] C. Upson, T. Faulhaber, D. Kamins, D. Laidlaw D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Computer Graphics & Applications*, 9(4):30–42, July 1989.

[18] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[19] John M. Vlissides and Steven Tang. A Unidraw-based user interface builder. In *Proceedings of the ACM SIGGRAPH Fourth Annual Symposium on User Interface Software and Technology*, Hilton Head, SC, November 1991.

[20] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, December 1992.

[21] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

A   Visualization Catalog

A.1   Allocation Matrix (Figure 1.4)

Purpose

Reveals which classes instantiate other classes.

Elements

Classes appear along the left and bottom edges. Classes are topologically sorted so that subclasses always appear above or to the right of base classes. Each square in the matrix denotes the number of instances of a class on the bottom that are allocated by a class on the left. A square's color reflects how many instances have been allocated. The color key at the bottom indicates the mapping of number of instances to color, from few (reds) to many (dark violets).

Interpretation

- **Static:** The view shows the relative number of allocations among classes.

- **Dynamic:** Color changes indicate the time at which classes allocate instances and the rate of allocation.

Related Visualizations

Patterns in this view are related to call patterns in the inter-class call matrix (A.6), because normally a class will communicate with classes it instantiates.

A.2   Class Time Chart (Figure 1.6)

Purpose

Presents method activation over time, grouped by class.

Elements

Classes are arranged along the left edge. They are sorted topologically so that subclasses always appear above base classes. Classes appear along the left edge as they are first instantiated. The horizontal axis represents time. At each step in time, a colored square is drawn for each class. Squares are color-coded as follows:

- **green** for classes having at least one instance but no active method

- **red** for the class of the currently active instance

- **blue** for the class of the currently active method if it is operating on an instance of a derived class

- **violet** for classes having methods on the call stack

Interpretation

- **Static:** A program that exploits code inheritance extensively will show many blue squares over time. Long horizontal bands of red indicate a class with heavily used method(s), thus representing potential execution hot-spots.

- **Dynamic:** Many new classes appearing along the edge of the view may indicate a new execution phase.

Related Visualizations

Violet squares correspond to the call stack path of the inter-class call cluster (A.5). Blue squares are equivalent to off-diagonal squares in the intra-class call matrix (A.7).

A.3   Functions-Instances Matrix (Figure 1.7)

Purpose

Shows invocations of individual methods on individual instances.

Elements

Methods appear along the left edge as they are first invoked; instances appear along the bottom edge as they are created. Each square in the matrix denotes the number of invocations of the method at the left on the instance at the bottom. A square's color reflects the number of invocations. The color key at the bottom indicates the mapping of number of invocations to color, from few (reds) to many (dark violets).

Interpretation

- **Static:** Instances showing constructor calls without corresponding destructor calls suggest memory leaks. Instances showing *only* constructor and destructor calls may reflect copy constructors or unaccessed (and potentially unneeded) objects.

- **Dynamic:** Many new instances and/or methods appearing along the edges of the view may indicate a new execution phase. Noticeable color changes draw attention to popular instances and methods.

Related Visualizations

The functions-instances matrix can be used in conjunction with class-based visualizations such as the inter-class call matrix (A.6) to investigate behavior at a finer granularity.

A.4 Histogram of Instances (Figure 1.3)

Purpose

Shows instances grouped by class and indicates their level of activity.

Elements

Classes are arranged along the left edge. They are sorted topologically so that subclasses always appear above base classes. Classes appear along the left edge as they are first instantiated. Each square in the view denotes an instance of the class to the left. A square's color reflects how many calls have occurred on the instance. The color key at the bottom indicates the mapping of number of calls to color, from few (reds) to many (dark violets). A white square represents an instance that has been deleted; such squares will be reused by new instances.

Interpretation

- **Static:** Classes with similar or identical bars suggest close coupling or containment. Unexpectedly large numbers of instances may suggest a memory leak.

- **Dynamic:** Instances from different classes appearing and disappearing in unison are another indication of close coupling or containment. Bars that grow rapidly can reflect object creation in tight loops. Squares flashing rapidly between red and white indicate short-lived objects, which may suggest excessive copy constructor calls.

Related Visualizations

The class time chart (A.2) shows a history of instance activity.

A.5 Inter-Class Call Cluster (Figure 1.1)

Purpose

Provides a dynamic overview of how objects communicate by displaying classes spatially according to communication frequency.

Elements

Classes appear as floating labels. The more two classes communicate, the closer they will appear; classes that do not communicate repel each other. Classes having an instance with a method on the call stack are connected by blue lines. A red line leads to the class with the currently active method.

Interpretation

- **Static:** Clustered classes are likely to be tightly coupled and/or part of the same subsystem.

- **Dynamic:** Concentrated activity of the red line for long periods between elements of a cluster suggests an execution hot-spot. Any of the following phenomena may indicate a new phase in the program's execution: (1) many new classes bursting out of the center and coalescing into clusters; (2) a drastic change in the shape of the call stack path; and (3) major shifts in the positions of class labels.

Related Visualizations

The inter-class call matrix (A.6) provides a cumulative record of communication patterns.

A.6 Inter-Class Call Matrix (Figure 1.2)

Purpose

Provides a cumulative overview of object communication summarized by class.

Elements

Classes are arranged along the left and bottom edges. They are sorted topologically so that subclasses always appear above or to the right of base classes. Classes appear along the edges as they are first instantiated. Each square in the view denotes the number of calls to methods of the class on the bottom by methods of the class on the left. A square's color reflects how many calls have occurred. The color key at the bottom indicates the mapping of number of calls to color, from few (reds) to many (dark violets).

Interpretation

- **Static:** The ordering of classes along the axis indicates their instantiation order. Vertical bands above the diagonal may reveal a base class that is heavily used, either through inherited code or by explicit calls from subclasses. Horizontal bands suggest classes that drive or contain instances of many others. Squares on the diagonal indicate calls

to self. Clusters near the diagonal are a sign of classes that are instantiated together and are designed to work together. Dark areas also indicate closely-coupled classes.

- **Dynamic:** Many new classes appearing along the edge of the view may indicate a new execution phase. Noticeable color changes draw attention to classes that interact heavily.

Related Visualizations

The inter-class call cluster (A.5) reveals more of the dynamics of inter-class communication. The *intra*-class call cluster (A.7) can verify heavy base class use.

A.7 Intra-Class Call Matrix (Figure 1.8)

Purpose

For each class, shows which class' methods are invoked on its instances.

Elements

Classes are arranged along the left and bottom edges. They are sorted topologically so that subclasses always appear above or to the right of base classes. Classes appear along the edges as they are first instantiated. Each square in the view denotes the number of times a method of the class on the bottom operated on an instance of the class on the left. A square's color reflects how many calls have occurred. The color key at the bottom indicates the mapping of number of calls to color, from few (reds) to many (dark violets).

Interpretation

- **Static:** A square on the diagonal signifies an instance operated upon by a method of its own class. Squares above the diagonal correspond to invocations of base class methods on derived class instances. A square below the diagonal would indicate invocation of a method from a class instantiated *after* the receiving instance, which is impossible because the classes are sorted topologically; thus this view must always be upper-triangular.

  A program that exploits code inheritance extensively will show many squares above the diagonal. Vertical lines indicate classes with methods inherited from a common base class.

- **Dynamic:** Many new classes appearing along the edge of the view may indicate a new execution phase. Noticeable color changes draw attention to heavy interaction between related classes in the inheritance hierarchy.

Related Visualizations

The class time chart (A.2) also reveals extensive use of code inheritance.