Visualising the Structure of Object-oriented Systems

Trent Hill*, James Noble**, John Potter***.
* Macquarie University, Sydney, Australia.
** Victoria University of Wellington, New Zealand.
*** University of New South Wales, Sydney, Australia.
{thill, kjx, potter}@ics.mq.edu.au

Abstract

Executing object-oriented programs have a complex structure consisting of numerous objects connected by interobject references. This structure, called the program's object graph, is hard to understand, and this complicates learning, teaching, debugging and maintaining object-oriented programs. While visualisation can be used to display object graphs, the size and complexity of typical object graphs also makes visualisation difficult. We have developed ***ownership trees*** *as simple yet powerful method of extracting a program's implicit encapsulation structure from its object graph. We present a program visualisation tool that makes use of ownership trees to display the structure of object-oriented programs. Our prototype implementation suggests that visualisations based on ownership trees can effectively display the structure of fifty to sixty objects in a program simultaneously.*

1. Introduction

The run-time topology of an object-oriented program is a collection of objects interconnected by inter-object references —the program's *object graph* (see Figure 1)**.** A typical object-oriented program creates a large number of objects and a large number of references between them, so its object graph is a large and complicated structure. As a program executes, the topology of its object graph evolves over time: new objects are created and join the object graph; old objects are dereferenced by the program and leave the graph; and assignments to objects' variables add, delete, and change the graph edges.

An object graph describes, in essence, the "shape" of an object-oriented program. As the techniques of object-oriented programming have matured, the design of structures of interacting objects has come to be seen as at least as important as the design of the objects themselves (and their classifications using inheritance), if not moreso [27]. The *Design Patterns* textbook [7], for example, explicitly encourages programmers to "*Favor object composition over inheritance*", that is, to build and adapt programs by designing and adjusting the dynamic structure of program's object graphs rather than the static

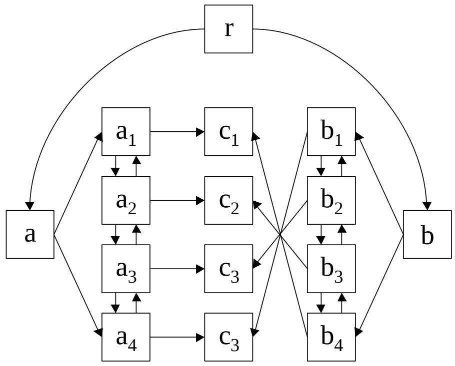

Figure 1. A Simple Object Graph. This object graph has root *r* and two doubly linked lists *a* and *b*. *a* and *b* have associated link objects a_i and b_i respectively for i = 1, ..., 4. *a* and *b* both contain the same data (c_1, c_2, c_3 and c_4) but in reverse order.

inheritance hierarchies between classes defining those objects.

Programs' object graphs are large and complex. Even a small-to-medium-sized object-oriented program may contain several tens of objects, while a large program may contain millions, if not billions. The use of design patterns and favoring composition over inheritance only makes programs' object graphs larger and more complex, as they reduce the number of classes and the complexity of program's inheritance hierarchy by increasing the number of objects and the complexity of their interrelationships.

Understanding the structure of a program's object graph, then, is a crucial prerequisite to understanding the program itself, and thus to debugging, maintaining or extending the program, or even simply to learning how it works. Software visualisation techniques have been fruitfully applied to many other aspects of object-oriented programming (and programming in general, of course) [5,26]. Unfortunately, the size and complexity of a dynamic object graph makes effective visualisation challenging. Creating a layout that allows a user to extract meaningful information from such a large structure is difficult using traditional graph layout algorithms, so most existing approaches avoid the problem, displaying only a few objects at any time, and requiring programmers to explicitly navigate around the object graph.

The aim of the work described in this paper is to investigate the use of software visualisation techniques to present larger object graphs to programmers. We adopted three main principles to guide our investigations. First, to support just the kinds of programs written by second year university students a visualisation would need to display medium-sized object graphs containing at least fifty to one hundred objects. Ideally, programmers should be able to see fair-sized portions of a program's object graph in one display, rather than having to navigate through the program one object at a time. Second, since programmers debugging their programs could not be expected to lay out fifty or sixty objects, the visualisation layout would have to be automatic, and could not depend on programmers or users supplying detailed semantic information about the structure of their programs. Third, to provide as much assistance to programmers as possible, the layout itself should provide information about the semantic structure of the objects in the program.

In this paper, we present a novel technique for dynamically visualising the structure of object graphs that meets these three goals. The key contribution of this visualisation is that it uses *object ownership trees* [21] to guide the layout of the objects to be displayed. Ownership trees extract implicit structural features of an object graph that correspond to the important principles of object-oriented design, such as encapsulation, abstraction, modularity, and information hiding [4]. In this way, ownership tree visualisation automatically organises the display of an object graph to reflect the conceptual structure of the underlying program.

We have implemented a prototype ownership tree visualisation tool to display the objects found in Java programs [8]. The prototype is written in Java, and uses a modified Java virtual machine to gather information about the object graph of a running program, which is then displayed [11]. Although the prototype system is limited in performance, we observed that the ownership tree visualisations can display the structure of quite large object graphs, without requiring programmers to lay out the displays or provide application-specific information about their programs.

The rest of this paper is organised as follows. We begin by briefly describing the properties of an object graph, and how ownership trees can capture their implicit structure. In section 3 we extend the ownership tree to a stack based approach appropriate for Java object graphs, and in section 4, we describe how we use ownership trees to lay out Java object graphs. Finally we discuss some related work and summarise the paper.

Further information about ownership trees in general, and our various attempts to use them for visualisation is available in our earlier papers [21,8,23]. This paper provides an in-depth description of the final visualisations we have constructed, and attempts to place ownership tree visualisations within a wider context.

2. Object Graphs and Ownership Trees

An *object graph* for an executing object-oriented program is a directed graph where a vertex represents each object in the system and each interobject reference is represented by an edge. A distinguished *root* vertex represents the root object in the program. An object is *alive* if it is reachable from the root; if not, the object is considered *garbage* and is excluded from the object graph.

Given an object graph G and a root vertex ρ, we say a vertex α *owns* a vertex β if and only if every path from ρ to β includes α [21]. The ownership relation gives a simple definition of the *interior* and *exterior* of an object. The interior of α is the set of objects α owns — that is, all objects that would be garbage if α was deleted. The *exterior* of α is all objects that are not α and are not in the interior of α. Since the ownership relation is transitive [21] the ownership relation forms a hierarchy on the graph. Each object has a unique *immediate owner*. An object's immediate owner is its immediate superior in the hierarchy. This hierarchy can be represented as the *ownership tree* with root ρ, a node for each vertex in the object graph, where every node's parent is its immediate owner.

In Figure 1, for example, the root r owns all other vertices trivially since every path from the root includes the root. Each list owns their respective link objects because each path from the root to the link objects includes the list object. Since the data c_i is accessible from both lists, it is not owned by either and is *promoted* to the root. In this manner the ownership tree in Figure 2 may be constructed.

In graph-theoretic terminology object's owners are known as their corresponding vertices *articulation points* or *dominators* [14]. Dominators and dominator trees are widely used to analyse control flow graphs in compilers; Potter, Noble and Clarke [21] recognised the use of ownership to identify structure in programs' object graphs.

An ownership tree allows us to impose a tree structure on the object graph, easing the problem of graph layout for visualisation. Since an ownership tree comprises all vertices reachable from the root, it will include all live objects in the system in the layout. More importantly, the properties of the ownership tree structure have important semantics when applied to an object graph.

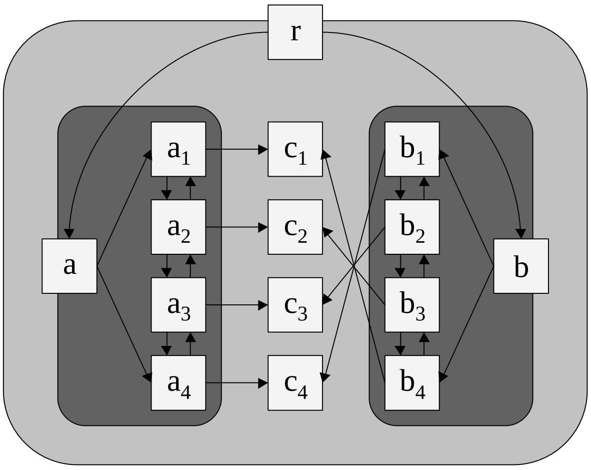

Figure 2. Ownership Tree. The root *r* owns *a, b,* and c_i, while *a* owns a_i and b owns b_i.

2.1 Properties of Ownership Trees

An ownership tree of a graph with root ρ has the following properties:

Reachable Vertices. Since all vertices reachable from the root are owned by the root, all reachable vertices are part of the tree. Conversely, if a vertex is not reachable from the root, there will be no paths from the root to that vertex so that vertex will not have any owners. So if a vertex is not reachable from the root it will not be part of the ownership tree. Therefore the ownership tree traverses the entire object graph and automatically excludes garbage.

Valid Edges. If there is a reference from α to β in the underlying graph, then α is in the interior of β [21]. In other words, a reference may only go to

- a child;
- an ancestor; or
- the immediate child of an ancestor.

Note that there is not necessarily an edge in the object graph between a parent and a child node in the ownership tree. For example, in figure 2 *r* is the parent of c_1 but there is no edge between *r* and c_1.

2.2 Ownership Trees and Encapsulation

Encapsulation is a fundamental concept in object orientation [4]. An object is usually an aggregate, implemented by of several other objects that comprise the internal representation of the aggregate. Encapsulation restricts access to these inner objects by unrelated objects, usually by not allowing unrelated objects to reference internal structure — we say that the inner objects are contained within the aggregate. Containment is important because it supports several principles of object-oriented design, such as abstraction, information hiding, and modularity [4]. The aggregate object can present an abstract interface to the rest of the program, can hide information within its implementation, and can be reused or replaced as a single modular component precisely because its inner implementation objects cannot be accessed from outside.

The notion of encapsulation maps directly to ownership in the object graph [21]. By not allowing references to inner objects, they will become the interior of the aggregate object in the ownership tree. References from an object's exterior cannot refer to an object's interior, so interior objects must be hidden from outside. In this way, an ownership tree reveals the natural containment relationships that are implicit in an object graph.

2.3 Ownership Trees and Sharing Scope

An object is *aliased* if it is referred to by more than one object [9]. Aliasing means that an object's internal representation can be directly referenced and hence changed without using that object interface. If an object does not expect such behaviour, it may be forced into an inconsistent state. Unrestricted aliasing results in programs that are hard to understand, maintain, and analyse [21, 16, 9].

If an object is aliased then there is a reference to it from more than one object. In an ownership tree, an aliased object cannot be owned by more than one object, so the aliased object must be *promoted* to a more senior position in the ownership tree — that is, closer to the root than any of the objects that refer to it. The *sharing scope* of an object is the object's level in the ownership hierarchy. Hence, an ownership tree provides information about the level of aliasing of the objects that it describes.

2.4 Dynamic ownership trees

The *valid edges* invariant of an ownership tree means that a reference in the underlying graph may only go to a child, an ancestor, or the child of an ancestor. As the object graph evolves, and new references are made to existing objects, we expect this invariant to be broken. When a new reference does not obey this invariant, the ownership tree must be restructured. Alstrup and Lauridsen [1] give an efficient algorithm for restructuring the ownership tree after a change in the underlying graph.

In the implementation we have chosen, during this restructure objects are again promoted in the tree but never demoted. The reason for this is that we want to see the overall sharing scope of an object, not just its current scope. If an internal object has been exposed to an external object, then that object could have changed the state of the internal object, whether or not the external object currently holds a reference to the internal object. In this way, the dynamic ownership tree displays a “worst-case” perspective, accumulating past ownership information as well as the illustrating the structure of the current object graph.

3. Ownership Trees and Java Applications

Unlike Eiffel and Smalltalk, Java does not have a single root object. To handle this, we have developed a model where Java ownership trees are rooted at stack frames rather than objects. We say that each stack frame on the call stack can hold references to objects. These references are created by instantiating local objects, passing objects through formal parameters to a method or returning objects from methods. The result is that rather than having a single ownership tree to model the program, we have a stack of ownership trees.

3.1 Interaction between stack frames

Older stack frames are deeper frames in the call stack and *younger* stack frames are the more recent call stack frames. Older stack frames and their interiors are more senior in the ownership hierarchy than younger stack frames.

Stack frames are not independent. References between stack frames can originate directly from references to formal parameters, returning values or by passing references back through formal parameters. To deal with this we allow references from objects owned by younger stack frames to objects owned by older stack frames (since a valid reference may go to an ancestor). These references may result from obtaining references through the formal parameters of a method.

A reference from an older stack frame to a younger stack frame requires the ownership tree to be restructured. The reason for this is that the younger object is now accessible directly from an object than is more senior in the ownership hierarchy. The young object and other objects accessible from it are *promoted* to the same ownership tree as the older, referring object. Promotions occur due to objects being returned from a function, or via manipulation of formal parameters.

As the stack grows and shrinks, we expect to see structure being formed in the younger stack frames and either being discarded or passed back through such restructuring to be integrated with the older structure.

3.2 Stack based ownership trees in a multi-threaded environment

Since Java supports multi-threaded applications, there may be any number of stacks operating concurrently. We say that each stack owns all of the frames created on it. An object may be aliased by more than one stack, however, either by passing a reference into a thread when the thread is created or by providing a global point of access, such as a static variable.

When an object can be accessed by more than one stack, it can no longer be owned by any of them since there are paths to it from each stack. This object is therefore promoted to a global ownership tree called the *heap*. The ownership tree rooted at the heap is senior in the ownership structure to all of the stacks so, as with the stack frames, references may go from an object owned by the stack into the heap, but a reference from an object owned by the heap to an object owned by a stack will cause the stack object to be promoted.

Figure 3 shows the final ownership structure for multithreaded Java programs.

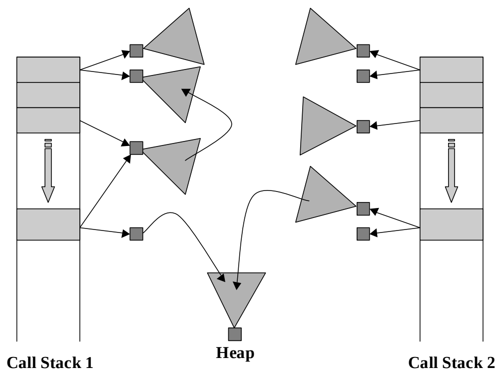

Figure 3. Multithreaded Ownership. Two threads have their own stacks of ownership trees, and a global heap.

4. Visualising Java object graphs

An effective visualisation must provide concurrent views of the ownership tree and the object graph of multiple stack frames on multiple stacks. Attributes of a good graph layout include minimising edge crossings and bends, minimising total layout area, finding layouts with short edges or emphasising symmetry [17]. Meeting such requirements efficiently is an open research problem. Due to the expected size of the graph being visualised, we have concentrated on minimising total layout area. In addition, the layout of a dynamic graph must be fairly stable. As structure is built and promoted, it should be obvious exactly what is being promoted and unrelated structure should not be visibly affected.

We have designed and implemented a visualisation for object graphs based on ownership trees. Our layout for a single ownership tree rooted at a stack frame is shown in Figure 4. The object graph corresponds to that shown in Figure 1. This visualisation exhibits a more obvious tree structure and requires significantly less space than the layout in that figure.

In the ownership tree visualisation, rectangular icons represent *objects*; each object can be labeled with an individual name or the name of the class to which it belongs. An *ownership bar* extends horizontally from the top of every aggregate object, that is, every object that owns at least one other object. The interior of an object (that is, the objects that are encapsulated within that object) is displayed under their owner's ownership bar.

Arrows between vertices represent interobject *references*. We noted earlier that a reference may go to a child, an ancestor or the child of an ancestor. An arrow pointing from or to an ownership bar denotes references to a child or an ancestor. In the example, *r* refers to {*a*, *b*}, *a* refers to {a_1, a_4} and *b* refers to {b_1, b_4} in this manner. We call such references *inline* references.

References to the child of an ancestor cannot be displayed inline and so extend below the tree. To show such references, we introduce *anchor* symbols. An anchor for an object α is a global point for references whose destination is α. In the visualisation, an anchor for

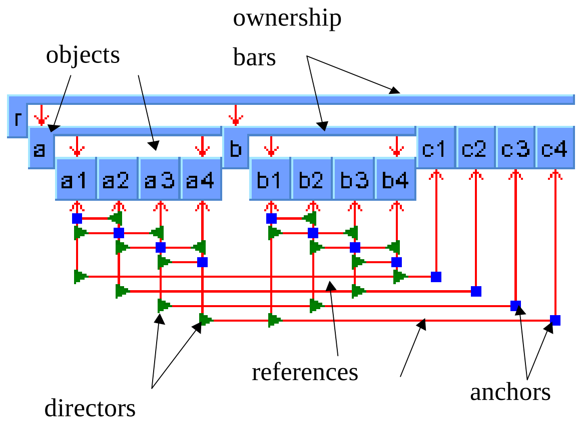

Figure 4. Ownership Tree Layout

an object is represented by a small square some distance directly below that object. If an object β refers to α and cannot use an inline reference, the visualisation will show a corresponding arrow from β to the anchor of α by first moving vertically to the height of the anchor, then moving horizontally to the anchor. In the example, a_1 refers to {a_2, c_1} in this manner. We call these references *baseline* references. An anchor is allocated to any object than has an incoming baseline reference from any other object.

We also use *director* symbols. A director is a small triangle placed at the intersection of the vertical line from the source and the horizontal line to the anchor. The director avoids ambiguity as well as indicating the direction of the anchor.

4.1 Allocating anchors

The main challenge with this layout is how to determine an appropriate height for the anchor of an object. This height must be chosen so that the source and destination of each reference is unambiguous. Ambiguity occurs when the horizontal line to two anchors at the same height overlap. Two objects may have anchors at the same height but references to one must not overlap references to another. We considered three strategies for allocating the anchor height of an object during an in-order traversal of the tree:

Unique height. We first considered allocating each object with an inward baseline reference a unique anchor height. This scheme is efficient and easily implemented, but it is inappropriate for a large visualisation. Using this approach, the anchor height will become roughly proportional to the distance covered in the traversal, resulting in a diagonal layout. An earlier object tree visualiser used this algorithm [23].

Best fit. Another option is to study the anchors and their corresponding references and determine the smallest possible anchor height that does not cause ambiguities. In this implementation, each object would need to be able to compute the set of spaces that have not already been used by a reference to another object. This strategy is much more efficient in terms of space usage, however implementation would be computationally expensive.

Partial fit. The partial fit approach is a compromise between the first two approaches, as was adopted in our prototype. Instead of keeping track of every free space available as in best fit, each object keeps track of the point above which all spaces are free. This point is called the *free line*. With this information, we can find a reasonable anchor for a vertex.

Initially, the free line for each object is initialised to be just below that object. To calculate the anchor height for an object α, we first compile the set of objects that have an baseline reference to α unioned with α. From this set we find the first and last object in an in-order traversal of the tree. The anchor height for α is then the maximum free line between the first and last object in an in-order traversal of the tree. After allocations, the free line for each object between the first and last object is updated to be just below this anchor height.

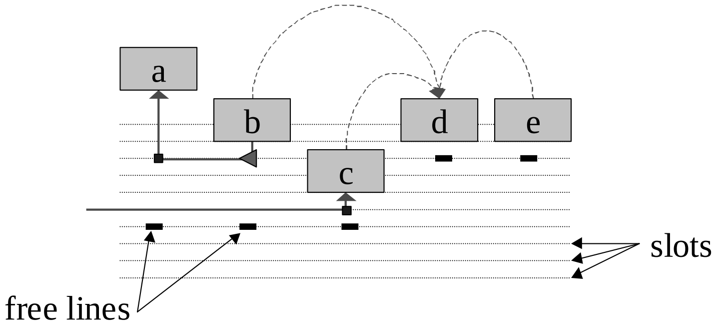

Figure 5. Allocating Anchor For d

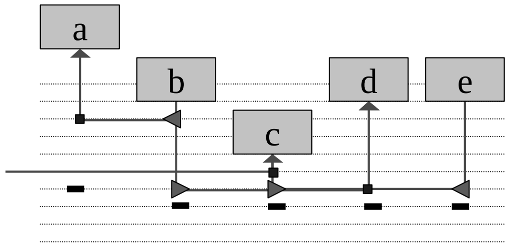

Figure 6. Resulting Anchor

In Figure 5, the free line for each node is shown as a thick line below the node, and anchors have been allocated for nodes *a*, *b*, and *c*. Since anchors are allocated during an in-order traversal, *d* is next to be allocated an anchor. Nodes *b*, *c*, and *e* refer to *d*, and each requires a baseline reference. To calculate the anchor height for *d*, we determine the set of objects that have a baseline reference to *a* unioned with *a*. From this set we find the leftmost and rightmost nodes in the display, in this case b and e respectively, and the anchor height for a is then the lowest free line between these two nodes. After the anchor is allocated, the free line for each object between the leftmost and rightmost object is updated to be just below this new anchor height, as shown in Figure 6.

4.2 Stack-based Ownership

This design extends well visualising multiple stacks and threads (see Figure 7). Each stack frame is just treated as another child of a stack and references between frames can be implemented as baseline references. Each thread in the program will have its own stack. The stacks are placed underneath each other in a column. References between stacks need not be considered, because such shared objects are transferred to the heap.

The heap consists of a single ownership tree. References into the heap may originate from any object in any stack. There are no references from the heap to an object on a stack because such a reference promotes the stack object into the heap. A vertical ownership tree to the left of the stacks represents the heap. References into the heap cannot be shown at present but strategies to visualise these references are currently being considered.

4.3 Visualising large applications

The ownership tree visualisation was designed to give a detailed view of the object graph and ownership tree in

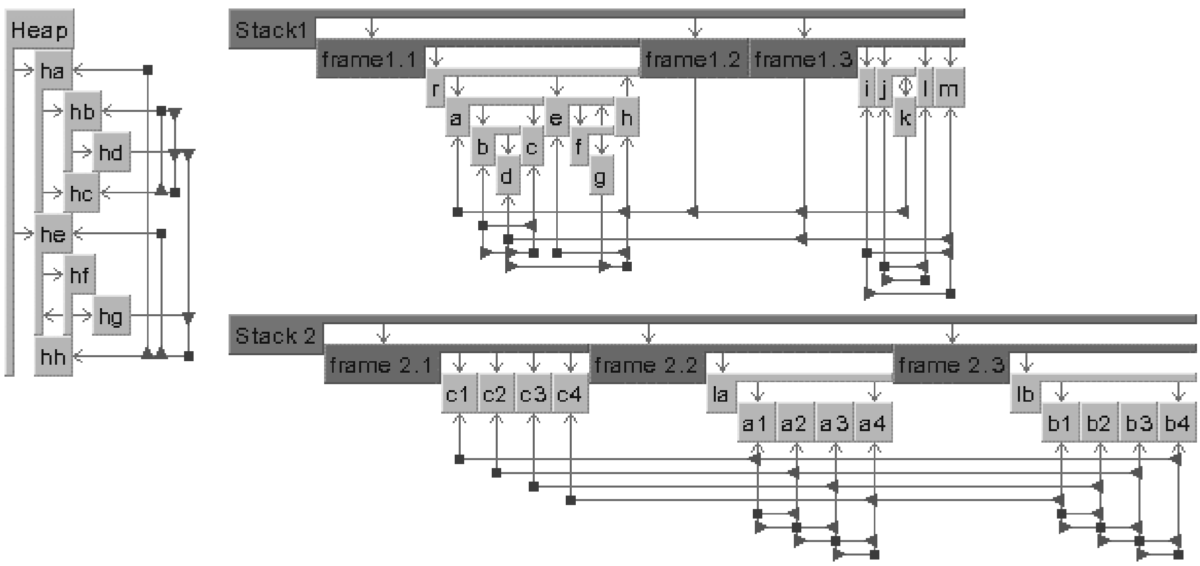

Figure 7. Multithreaded Layout

a minimal amount of space; however the amount of detail it presents is overwhelming for large applications. A user will typically want to concentrate on a certain section of the graph in detail but still have an overall idea of the general structure elsewhere.

Because the ownership tree is build upon containment, the ownership structure below an object is the interior or internal representation of that object, and there can be no references to these objects from outside their owner. The prototype allows users to hide the interior of an object on demand by *collapsing* an object.

When a object is collapsed, all references to or from a child will be visualised as if it were a reference to or from the visible ancestor of that child. Such references are called *indirect references* and are displayed using a different colour, as shown in Figure 8.

5. Related Work

The general success of object-oriented programming, combined with a general acknowledgment of its complexity, has inspired a large number of software visualisation systems designed to illustrate various aspects of object-oriented programs.

Perhaps the most common systems (certainly, the only ones that are widely used) are CASE tools and programming environments that display static class and inheritance hierarchies, as presaged by systems such as FIELD [22]. Concentrating purely on program's static aspects, these systems do not even attempt to display programs' object graphs.

The first systems to display objects and their dynamic interrelationships (rather than just classes) were graphical debuggers that produced simple views of objects and the references between them: the Smalltalk Program Explorer [3] is one example that actually gained some commercial use, the IBM program explorer and the Self programming environments were two research systems [12,13,25]. As their names suggest, these systems allowed the user to "explore" or navigate through their programs, selecting one object at a time to add to the visualisation, and positioning that object manually: in general this approach is only feasible for visualising around twenty objects. Systems such as Incense [15] and the Pascal Genie [6] produced similar displays of records and pointer structures even before object-orientation became widespread.

More recently, several visualisations have been designed to display the behaviour of large numbers of objects simultaneously. Systems such as the Object Visualizer, the Interaction Scenario Visualizer and J-Insight can display details of object creation and deletion, method invocation, and analyse recurring patterns within programs' execution traces [10,.18,19]. The more flexible of these systems also require programmers to annotate programs in advance, describing the abstractions they contain and identifying the implementation components whose behaviour should be monitored to produce the visualisations [24,28]. Most of these systems focus on programs' execution *behaviour* rather than their *structure* or object graphs. One notable exception is the Super-Jinsight tool, which can display objects referred to (or, for cross-reference) all objects referring to a selected object [20]. Like many behavioural visualisations, Jinsight can compress its displays by identifying common patterns of references among objects; however, the objects to be displayed are basically chosen via manual navigation through the object graph as in a graphical debugger.

In contrast to these systems, our ownership tree visualisation displays the *structure* of object-oriented programs, in particular, the containment or encapsulation structure of objects within programs; it is the only visualisation to present this information. As the ownership structure is implicit in programs' object graphs, the visualisation does not need users to annotate or navigate through programs, or manually select objects to view (although users can collapse nodes to save screen real estate as necessary). Our visualisation does not require manual layout, because the ownership structure is a tree and layout is quite straightforward.

6. Conclusion

This paper has presented an object graph visualisation based on ownership trees. Compared to previous

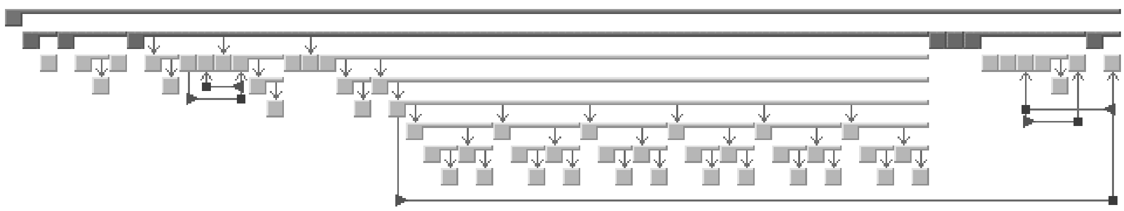

Figure 8. The Java Compiler

approaches, the ownership tree visualisation can display the structure of medium-sized object graphs, such as the excerpt from a Java compiler shown in Figure 8. The visualisation does not require any manual layout or navigation of objects in the program to produce a useful visualisation. By laying out object graphs based on their implicit containment relationship, the ownership-tree visualisation presents an important part of the structure of object-oriented programs.

References

[1] S. Alstrup and P. Lauridsen. *A Simple Dynamic Algorithm for Maintaining a Dominator Tree*, Tech. Rept., Dept. of Comp. Sci., Univ. of Copenhagen, Copenhagen, 1996.

[2] G. Battista, P. Eades, R. Tassmassia, and I. Tollis. *Algorithms for Drawing Graphs: an Annotated Bibliography*, 1994.

[3] K. Beck. *Pictures from the Object Explorer*. Technical Report, First Class Software Inc, 1994.

[4] G. Booch. *Object-oriented Analysis and Design With Applications*. Addison-Wesley, 1994.

[5] M. Burnett, A. Goldberg and T. Lewis, eds. *Visual Object-Oriented Programming.* Manning 1995.

[6] Chandhok et.al. *Pascal Genie*. Chariot Software Group.

[7] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns.* Addison-Wesley 1995.

[8] T. Hill, J. Noble, J. Potter. Visualising Implicit Structure in Java Object Graphs. In Proceedings of *SoftVis 1999 Workshop*. Sydney.

[9] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. *The Geneva Convection on the Treatment of Object Aliasing*, OOPS Messenger, 3(2), Apr. 1992.

[10] D. F. Jerding and J.T. Stasko and T. Ball. Visualizing Interactions in Program Execution. In the Proceedings of the International Conference on Sofware Engineering. (ICSE). IEEE Press. 1997.

[11] C. Laffra. *Advanced Java*. Prentice Hall, 1997.

[12] D. Lange and Y Nakamura. Interactive Visualization of Design Patterns can help in Framework Understanding. In *OOPSLA Proceedings*, 1995.

[13] D. Lange and Y. Nakamura. Object-Oriented Program Tracing and Visualization. *IEEE Computer, Vol.30 No.5, May 1997.*

[14] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[15] B. A. Myers. Incense: A System for Displaying Data Structures. In *SIGGRAPH Proceedings*, 1983.

[16] J. Noble, J. Vitek, and J. Potter. *Flexible Alias Protection*. In ECOOP Proceedings 1998.

[17] S. North. *Visualizing Graph Models of Software*, from Software Visualization, The MIT Press, 1998.

[18] W. De Pauw, R. Helm, D. Kimmelman, J. Vlissides. Visualizing the Behavior of Object-Oriented Systems. In *OOPSLA Proceedings*. 1993.

[19] W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman. Execution Patterns in Object-Oriented Visualization. In *COOTS Proceedings*. 1998.

[20] W. De Pauw and G. Sevitsky, Visualizing Reference Patterns for Solving Memory Leaks in Java. In *ECOOP Proceedings*, 1999.

[21] J. Potter, J. Noble, and D. Clarke. *The Ins and Outs of Objects*, Microsoft Research Institute, 1998.

[22] S. Reiss. *The FIELD Programming Environment.* Kluwer Academic, 1995.

[23] D. Santibanez. *Visualisation of Ownership Trees in Evolving Object Graphs*, Honours Thesis, Macquarie University, Sydney, Australia, 1998.

[24] M. Sefika, A. Sane, R. Campbell. Architecture-Oriented Visualization. In *OOOPSLA Proceedings*, 1996.

[25] R. Smith, J. Maloney and D. Ungar. The Self-4.0 User Interface. In *OOPSLA Proceedings*, 1995.

[26] J. Stasko, C. Patterson. *Understanding and Characterizing Program Visualization Systems*. Tech. Rept. GIT-GVU-91-19, Graphics, Visualization, and Usability Center, College of Computing, Gorgia Institute of Technology, Atlanta, 1993.

[27] A. Taivalsaari. Classes vs. Prototypes: Some Philosophical and Historical Observations. In *Prototype-Based-Programming.* J. Noble, A. Taivalsaari and I. Moore, eds. Springer-Verlag, 1999.

[28] R. Walker, G. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, J. Isaak. Visualizing Dynamic Software System Information through High-level Models. In *OOPSLA Proceedings, 1998.*