From Class Diagrams to Object Diagrams: A Systematic Approach

Paul V. Gestwicki, Bharat Jayaraman, Ashim Garg

[pvg|bharat|agarg]@cse.buffalo.edu

Department of Computer Science and Engineering
University at Buffalo, State University of New York
201 Bell Hall Box 60200
Buffalo, NY 14260-2000

December 13, 2004

Abstract

This paper addresses the problem of how to visually depict the object structures that arise during the execution of object-oriented programs. Such a need arises in systems for run-time visualization of object-oriented programs. A straightforward approach would be to treat the object structure as a directed graph and apply traditional graph drawing techniques. However, we show that such an approach results in suboptimal drawings since important structural information in the class diagram is overlooked. A more effective approach must utilize properties of the class diagram in guiding a traditional graph-drawing approach. In particular, objects that are instances of related classes should be drawn in proximity to one another. The contribution of this paper lies in providing a formal definition of two important properties that yield predictable object diagrams: recursive cluster and leaf cluster. While the former commonly occur in structures such as lists, trees, etc., the latter are motivated by the power law property of object graphs, namely, that there are very few objects with high degree and many with low degree. This paper shows that these properties can be efficiently detected in the class diagram, and also efficiently applied to produce aesthetic object diagrams. The techniques described in this paper form the basis for the rendering of object structures in the JIVE system for interactive visualization of Java programs.

1 Introduction

This work is motivated by the desire for better drawings of the object diagrams created by object-oriented programs. The execution state of a program can be described in terms of an object graph, where the vertices are objects and the edges are object references. Traditional techniques of graph drawing can be used to construct the object diagram, but important information about the objects would be lost. For example, each object is an instance of a class, and classes are related to each other. These relationships among classes are described by a system's *class diagram*, and they form a template of possible object structures. We propose that an effective drawing of object diagrams should reflect the system architecture as presented in its class diagram. To this end, we describe a novel technique that incorporates program-specific properties with graph-theoretic in order to more effectively draw object diagrams.

One of the most important aesthetic goals in drawing object diagrams is that instances of related classes are drawn in proximity to each other. The difficulty lies in defining formal methods of detecting and processing these relationships in a way that aids in drawing the object diagram. Our approach identifies two patterns in class diagrams that yield predictable object diagrams: *recursive clusters* and *leaf clusters*. Recursive clusters are formed by recursively-defined types; these commonly occur in data structures such as lists and trees as well as in hierarchical structures such as GUI frameworks. Leaf clusters are formed by the relatively simple data objects that are attached to larger aggregates. Our formulation of leaf clusters is based on the observation that the degree of objects in an object graph obeys a power law, where there are very few objects with high degree and very many objects with low degree [22]. We show how these two structures can be efficiently predicted from the class diagram, detected in the object diagram, and drawn in such a way that meets our aesthetic criteria.

1.1 A Motivating Example

A good example motivating the need for class diagram analysis in drawing object diagrams is provided by the binary search tree example. Figure 1 shows two designs for a binary search tree along with sample object diagrams. Figure 1(a) shows a class diagram for an implementation of integer binary search trees that encapsulates integer values within the tree nodes. This is a reasonable design in a language such as Java, where integers are represented with a primitive type rather than an object type [12]. Figure 1(b) shows an alternate design that stores the value of a node in a separate object.

Figure 1(c) shows a sample object structure for design (a), and parts (d) and (e) show two drawings of a sample object structure for design (b). The drawings in parts (c) and (d) have been generated using standard layered graph drawing techniques and aesthetics: topologically sort the nodes, assign them to layers, and reduce the crossings [4]. Diagram (c) emphasizes the hierarchical structure of the tree: each layer in the drawing represents a logical layer in the tree data structure. This highlighting of hierarchical structure is one of the benefits of layered graph drawing [26]. However, the drawing of part (d) lacks the clarity of (c). The underlying meaning of the structure is harder to determine from the visualization since `Data` and `BST` objects are treated homogeneously by the drawing algorithm. A better drawing of the same structure is found in part (e). This drawing emphasizes the tree structure, keeping each logical layer on a visual layer in the drawing, while the `Data` objects are more closely tied to their corresponding `BST` objects. This coupling in the drawing reflects the tight coupling of these two classes in the design, especially the topological dependency of `Data` objects upon their `BST` counterparts, where each `Data` is referenced by exactly one `BST`.

1.2 Aesthetic Goals

Effective graph drawing is achieved by balancing multiple conflicting visual aesthetic constraints. Managing these constraints is a combinatorially hard problem. We augment basic graph-drawing aesthetics, such as minimized total edge-length and minimized area, with aesthetic criteria that are specific to object diagrams. The overarching aesthetic goal is that related objects are drawn in proximity. Since "related objects" is vague, we specify the following concrete properties we wish to capture in our drawing:

A1 *Couple leaf objects with their aggregators.* We consider *leaf objects* to be those objects that do not reference other objects, and that are themselves referenced by exactly one *aggregator object*. We expect there to be a large number of leaf objects in an arbitrary program due to the power law distribution of object references: in normal programs, there are very few objects with a large number of references and there are very many objects with a small number of references [22]. This criterion is proposed as a technique to reflect the tight coupling of simple objects in program design with proximity of vertices in an object diagram.

A2 *Cluster recursive object structures.* The binary search tree of Figure 1 is a classic example of a recursive type: the tree is defined by repeated linking of objects of type `BST`. Recursive structures are widely used in programming as data structures, and their use is encouraged by several design patterns such as Composite and Decorator [7]. This criterion is proposed as a technique for highlighting the cohesiveness of a recursive structure. The intention is that objects that make up a recursive structure will be clustered into a topologically-constrained space of the drawing, which highlights their structural cohesiveness and conceptual grouping.

We propose that these aesthetic criteria will yield object diagrams that more closely match the architectural specification of the class diagram. In this document, we show that these properties can be detected and utilized in drawing by combining static class diagram analysis with dynamic analysis of object diagrams. These techniques can be incorporated into program visualization tools such as JIVE [10].

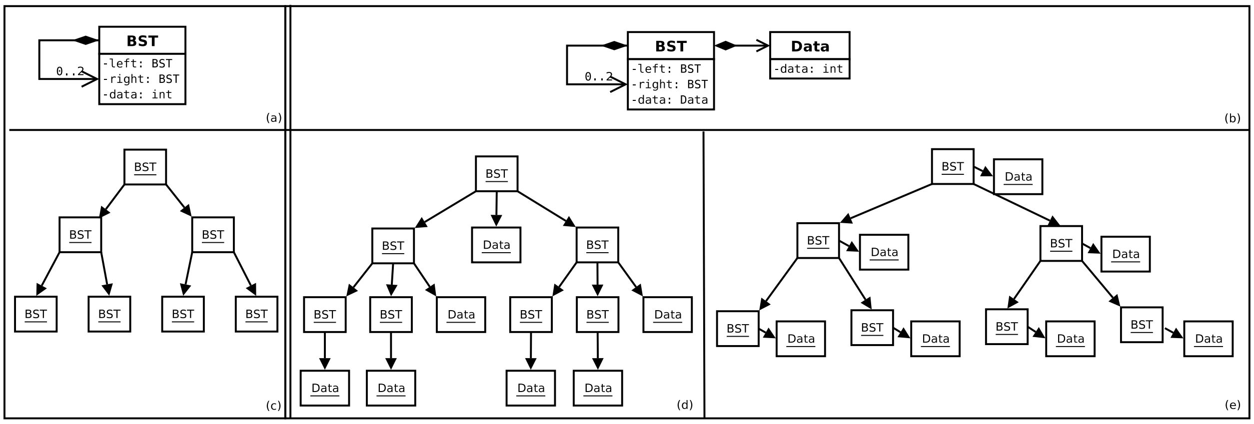

Figure 1: Binary search trees. Parts (a) and (b) show two alternate implementations for a binary search tree. Part (c) is an object diagram for the architecture in (a), and parts (d) and (e) show two possible object diagrams for the architecture in (b). Object diagrams (c) and (d) were drawn using standard layered graph drawing techniques and aesthetics. Diagram (e) was drawn by incorporating an analysis of the class diagram into the drawing.

1.3 Related Work

There has been significant effort in applying class-diagram analysis for drawing UML class diagrams themselves. These efforts are founded in the idea that a good drawing of a UML diagram is easier to understand [23]. Although these studies were performed on class diagrams, we expect that similar results would arise for other UML diagram types, including object and sequence diagrams [1]. GoVisual is a UML drawing tool that uses a clustering technique where clusters are formed based on generalization hierarchies [14]. It is one of several works that adopts a topology-shape-metrics to drawing UML class diagrams [6, 4], while other work focuses on using class diagrams as debugging tools [18]. However, none of these works have used class diagrams as a tool for generating object diagrams.

Many program visualization tools are designed to generate visualizations of large programs' behaviors over time. Bloom is an excellent example, as it provides a robust and flexible visualization system for viewing such runtime details as object allocations over time, call graphs, and performance data [24]. Other tools such as JGrasp and JIVE provide lower-level views of object state [17, 10]. The techniques we present would be more applicable to this type of visualization system.

Our drawing technique involves cluster detection and arrangement, but it is substantially different from existing techniques. Traditionally, clusters are either manually identified [8] or automatically detected in the graph [25]. These techniques are applicable to generic graphs that are domain-independent; the methodology described herein leverages the fact that object graphs are the manifestations of static class relationships.

The remainder of this paper is structured as follows: Section 2 provides formal definitions and background information on class diagrams and their processing; Section 3 presents our analysis of class diagrams and object diagrams and the resulting theorems; Section 4 demonstrates our technique on an example; and Section 5 presents conclusions and outlines future work.

2 Preliminaries

In this section, we describe class diagrams and the classes and relationships that comprise them. This includes an analysis of relationship annotations such as multiplicity and direction. We also examine the declaration of recursive types in class diagrams and their affect on object diagrams.

2.1 Class Relationships

At the level of the class diagram, all classes will be treated equally, with no differentiation between classes and interfaces. This allows a desirable level of language-independence, since the difference between these two idioms is language-specific. For example, interface and class declarations are separate and distinct in Java, but both are implemented as classes in C++.

Accepting this standard treatment for classes implies that UML generalization and realization relationships are similarly equivalent. A realization relationship from B to A means that B implements the method declarations of A; a generalization relationship from B to A means that B inherits the method declarations and attributes of A, subject to access-control modifiers. If we do not differentiate between interface classes (that is, classes with the ≪interface≫ stereotype) and non-interface classes, then we can treat realizations as generalizations without loss of generality.

Specific languages may have restrictions on what types of generalization are syntactically valid. For example, Java allows multiple inheritance of interfaces but only single inheritance of classes [12]. We will use the more general UML model, which allows multiple inheritance, with the understanding that specific implementations may either require customization or be able to exploit language-specific properties in order to increase performance.

The analysis of association, aggregation, and composition relationships is complicated by their ambiguity with respect to operational semantics. The UML describes these binary class relationships justifiably at the level of a design model [1]. Language-neutral modeling languages require ambiguity by definition, but this presents a discontinuity between a model and its implementation. The only clear way to uniquely map from a UML model to source code is to place restrictions on the modeling language [16]. Rather than narrowing the modeling language or accepting the ambiguities, we adopt the consensual definitions of binary class relationships as recently proposed by Guéhéneuc and Albin-Amiot [13]. The consensual definitions for association, aggregation, and composition, as they relate to model semantics, are summarized below.

Association An association from A to B in a model means that instances of A can directly send messages to instances of B.

Aggregation An aggregation is an association from A to B and is a relationship between a *whole* and its *part(s)*, respectively.

Composition A composition as an aggregation from A to B such that the instances of B are exclusively referenced by their corresponding instance of A. Furthermore, there is a lifetime dependency of B with respect to A such that if an instance of A is destroyed, its corresponding B objects are likewise destroyed.

These definitions assume that each binary class relationship is directed; this assumption is safe since bidirectional relationships can be expressed as a pair of directed relationships [20]. Association, aggregation, and composition form a hierarchical ontology of relationships: compositions are aggregations, and aggregations are associations. These relationships are likewise inherited through generalization, since attributes and methods are inherited, except when prohibited by access control modifiers or overriding and shadowing. Dependency is the weakest binary class relationship in the UML [1], and it has no necessarily manifestation in the object diagram. For the purposes of an ontology of relationships, we can consider associations to be dependencies.

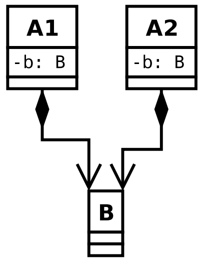

Figure 2: A class diagram showing two compositions to the same class. $A1$ and $A2$ are the "wholes" and B is the "part" in the compositions.

The presence of any of these relationships does not preclude the existence of other relationships. The exclusivity referenced in the definition of composition refers to the object-level, not the class-level. An example is shown in Figure 2; the fact that the relationships are compositions implies that the B object referenced by an instance of $A1$ will be different from the B object referenced by an instance of $A2$. In fact, this is a generalization of the property that the B object referenced by any $A1$ object must be different from the B object referenced by any other object.

Our analysis requires looking beyond binary class relationships to how these relationships are composed in class diagrams. To this end, we define two categories of relationship paths: generalization paths and association paths.

Definition 1 *A **generalization path** is a sequence of class identifiers* $\tau_1, \tau_2, \ldots, \tau_n$ *such that* τ_{i+1} *is an immediate superclass of* τ_i *for all .*

Definition 2 *An **association path** in a class diagram is a sequence of class identifiers* $\tau_1, \tau_2, \ldots, \tau_n$ *such that for any pair of types* (τ_i, τ_{i+1}), $1 \leq i \leq n-1$, *either:*

- *there is an association from* τ_i *to* τ_{i+1}*, or*
- *there is a generalization path from* τ_i *to* τ_i'*, there is a generalization path from* τ_{i+1} *to* τ_{i+1}'*, and there is an association from* τ_i' *to* τ_{i+1}'.

Association paths are defined in terms of generalization paths since associations are inherited by subclasses. An association path is generic: it can contain aggregations and compositions as well as "normal" associations. *Aggregation paths* and *composition paths* are defined similarly to association paths, substituting aggregations and compositions respectively.

2.2 Object Graphs

In JIVE, a custom visual notation is used for its capacity to clarify the semantics of object-oriented program execution [10]. For the scope of this paper, we will use a simpler notation, a syntax inspired by UML's object diagrams [1]. UML object diagrams are often used to indicate how messages are passed between objects, such as in collaboration diagrams. Our object diagrams are depict the state of objects as a snapshot of runtime, not the calling sequence or data flow.

Similarly, JIVE's notation for method execution depicts methods in their proper execution contexts [9]. We will elide method activations from our consideration of object structure. It is true that method activations may contain references to objects, and that this affects the object graph. However, the references in method activations are necessarily transient, existing only for the lifetime of the method activation.

Many programming languages, including Java, have a separation between static space and object space [9]. We ignore static contexts since their semantics are language-specific; however, we observe that since static contexts can have references to objects, static contexts can generally be treated as objects and included in the object graph. This is consistent with a Smalltalk-inspired treatment of classes as special types of objects [11].

Interclass relationships, as expressed in a class diagram, provide a template for the object graph that occurs at runtime. The nature of this correlation depends on the type of relationship. As mentioned previously, dependencies have no well-defined manifestation in the object graph unless the actual type of the relationship is more specific (association, aggregation, or composition).

Generalization. At runtime, inheritance hierarchies "collapse" into objects. It is possible to visualize the object in terms of its inheritance hierarchy [19], but generally the object can be treated as a single unit. Dynamic type checking will still recognize the object as inheriting from all of its superclasses.

Association. In order for an instance of A to send a message to an instance of B, it needs to obtain a reference to it. However, the presence of an attribute of type B in A would indicate an aggregation, according to the consensual definition. The reference to B must therefore be through a parameter to a method on A. The object reference from A to B is therefore a transient one, lasting at most as long as the method is active.

Aggregation. The implementation strategy for a part-to-whole relationship from B to A is to define attributes of type B within A [13]. This implies that in the object graph, instances of A will reference instances of B. Aggregation is not an exclusive relationship: these instances may be shared among other aggregate structures or may be bidirectional. The affect of multiplicity on aggregation is discussed in Section 2.3.

Composition. Composition is an aggregation, and so it is also implemented through object attributes. The lifetime dependency of composition relationships is a constraint on the behavior of the object graph, specifically that the part is exclusively referenced by the whole. Given a class A that composes B and objects a of A and b of B such that a references b, then a is the only object that references b, and when a is destroyed, b is also destroyed.

2.3 Multiplicity and Recursive Types

The multiplicity of a relationship in a class diagram has a direct impact on the possible object structures that the program can generate at runtime. By definition, given a relationship from A to B where the multiplicity for A is m and the multiplicity for B is n, then at runtime, there can be up to m instances of A in relationship with B objects, and n instances of B in relationship with A objects. We refer to these cases as *bounded multiplicity*. The asterisk (*) is used by convention to indicate *unbounded multiplicity*, where there is no theoretical maximum to the number of objects in the relationship. Since we are analyzing arbitrary states of program execution, we treat the multiplicity as a potential maximum of objects in relation, not a predicted number. When a program begins execution, for example, these relationships will have zero objects. In the dynamic environment of program execution, it is therefore sufficient to treat any multiplicity k as if it were $0..k$.

The multiplicity of aggregations in a class diagram also provides important hints about the implementation of the aggregation. In general, given an aggregation from A to B with multiplicity of $k \in \mathbb{N}$, this is implemented in A as either k attributes of type B or as an array of B with size k. Given an aggregation from A to B with unbounded multiplicity, the implementation must be through either a dynamically-sized array or a collection class.

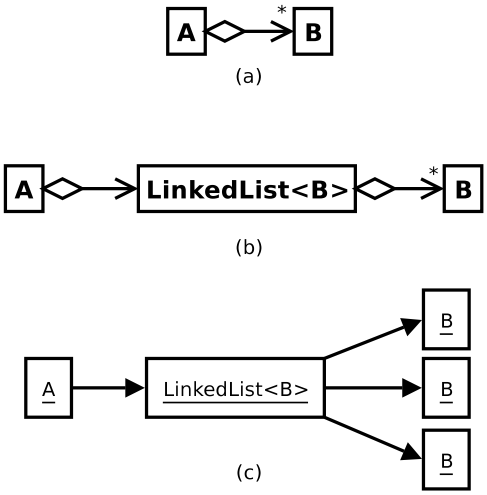

Figure 3: Aggregation realized as a linked list. Part (a) is a high-level class diagram, and part (b) shows how the aggregation in (a) is realized as a linked list in the implementation. Part (c) is a sample object diagram for the system.

Figure 3 gives a simple example in Java: part (a) shows an association with unbounded multiplicity; part (b) gives another view of the class diagram, this one closer to the implementation, highlighting the use of a `LinkedList` to realize the aggregation; and part (c) shows a sample object diagram for the aggregation. In the implementation of the `LinkedList` class, most likely there is a node class, such that the linked list composes nodes, and the nodes aggregate data objects (`B`, in our example). However, these are hidden in the class diagram, and so we have elided them from our object graph, as described previously.

The importance of recursive types in structured programming is evidenced by the many occurrences of recursive types in design patterns, including Decorator, Chain of Responsibility, and Interpreter patterns [7], in addition to their being standard fare in courses on data structures. Figure 4 shows an example of a recursive type, an instance of the Composite pattern, taken from the AWT package of the Java standard API [27]. `Container` is a subclass of `Component`, but a `Container` object aggregates `Component` objects. These classes and their subclasses are used to build graphical user interfaces. At runtime, the object structure is a tree that is rooted in the main application window and whose branches are made up of more `Component` subclasses such as `Label`, `TextField`, and `Button` objects.

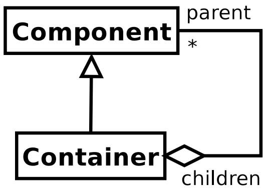

Figure 4: Sample recursive type from the `java.awt` package. The AWT/Swing API exemplifies recursive data types through the Composite design pattern.

The simplest recursive types are self-aggregating classes, that is, those classes C that have an aggregation to C. A classic example is the basic binary search tree, as shown in Figure 1. Those recursive types that aggregate their superclasses exhibit similar behavior to those that are strictly self-aggregating; this is the case in the `Component`-`Container` example of Figure 4. This leads to our definition of a *simple recursive class*:

Definition 3 *A* ***simple recursive class*** *is a class C that has an aggregation relationship with itself or with one of its superclasses. This aggregation is the class'* ***recursive aggregation****.*

Given a simple recursive type whose recursive aggregation has a multiplicity of one, there are three archetypical realizations of the aggregation in the object diagram, as shown in Figure 5 with example class

C. One possibility is that there will be only one instance of C, and it will bear a structural link to itself (Figure 5(b)). In the this case, there is no multi-object structure to consider. Another possibility is that there will be multiple instances of C, and they will be linked by a unidirectional series of object references, as shown in Figure 5(c). The structure of the objects in this case is quite disciplined. A third possibility is that there may be a chain of objects but with a single "back edge;" this case is shown in Figure 5(d) where the back edge points to the first node, but generally it can reference any one in the sequence. We can therefore assert that simple recursive types with multiplicity of one engender well-defined object structures, since in the worst case, there is only a chain of connected objects to consider, and in the simplest case, there is only one object.

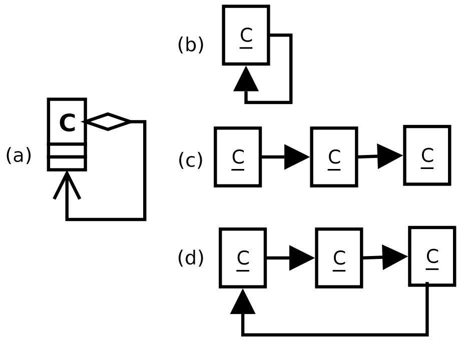

Figure 5: An example of a simple recursive class. Part (a) is the class diagram in which `C` is defined by a self-aggregation of multiplicity one. Parts (b), (c), and (d) are object diagrams showing some possible manifestations of the aggregation as object references.

Mutually recursive types involve a cyclic aggregation path among classes in separate inheritance hierarchies. When restricted to aggregations of multiplicity of one, the behavior of mutually recursive types is similar to that of simple recursive ones. An example of a mutually recursive type is given in Figure 6. In this example, A and B are classes stand in a bidirectional aggregation, shown in the figure as two unidirectional aggregations for clarity. Part (a) is the class diagram, and parts (b), (c), and (d) show possible manifestations of the association as object references.

For clarity when discussing recursive classes, we can define a general *recursive type*, but we first must formalize a *generalized aggregation path*:

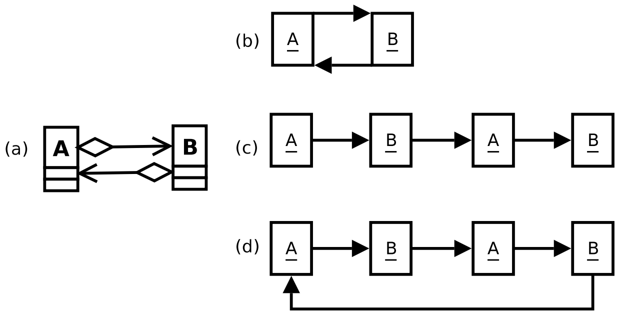

Figure 6: Mutually recursive types and possible object structures. Part (a) gives a simple mutually recursive type, and the rest of the diagram shows the patterns of object structures that are possible from it. Part (b) is a circular object structure, part (c) is a chained object structure, and part (d) combines the two into a circular chained object structure.

Definition 4 *A **generalized aggregation path** is a sequence of class identifiers $c_1, c_2, \ldots, c_n$ such that for any subsequence $c_i, c_{i+1}, \ldots, c_j$, one of the following is true:*

- *$i - j = 1$, and c_i aggregates c_j.*
- *c_{i+1} is an immediate superclass of c_i, and there is a generalized aggregation path from c_{i+1} to c_j.*
- *c_{j-1} is a superclass of c_j, and there is a generalized aggregation path from c_i to c_{j-1}.*

Definition 5 *A **recursive type** τ is made up of a set S of classes such that either:*

- *$|S| = 1$ and the class $C \in S$ is simple recursive.*
- *$|S| > 1$ and for any pair of classes $C_1, C_2 \in S$, there is a generalized aggregation path between them.*

A generalized aggregation path is an aggregation path that traverses generalization relationships. A generalized aggregation path from C to D can manifest in the object graph as a path of edges from an object of C to an object of D. The definition of a recursive type then is a formalism of the informal description provided above. We refer to any class diagram containing a recursive type as being a *recursive class diagram.*

Figure 7(a) provides an example recursive type that involves both generalization and aggregation.

Given an instance of C1, it references an instance of C2, which inherits a reference to an instance of C4, which inherits a reference to an instance of C1, thereby producing a cycle. Figure 7(b) shows a deceptively similar class diagram for a type that is possibly, but not necessarily, recursive. Given an instance of C6, it may contain a reference to an instance of C7 or of C8, in keeping with the rule of subtype polymorphism. This holds as well for C8, C9, and C10. If the first reference was to an instance of C8 and the second to C10, then there is a reference to C6, which would create a cycle. However, we cannot determine through analysis of the class diagram whether or not these conditions will be satisfied. Additionally, we observe that the implementation of this model as a recursive structure would require coercion, which is generally unsafe. We therefore do not consider the classes C6–C10 to form a recursive type. This example illustrates a fundamental complication of mutually recursive types: there are cases where it is impossible to determine through class analysis whether the object structure resulting from a class diagram will contain recursive elements or not.

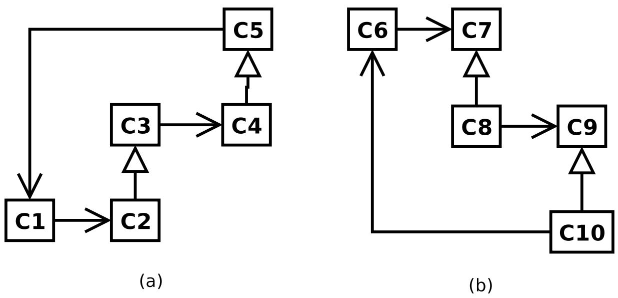

Figure 7: A pair of class diagrams containing cycles. Part (a) exhibits recursive types: classes C1, C2, and C4 are all necessarily part of a recursive structure. Part (b) shows a similar structure, but it does not necessarily contain recursive types.

3 Analysis

Now that we have explored binary class relationships, multiplicity, and recursive types, we can integrate them to make some observations and assertions about the relationship between class diagrams and object diagrams. Specifically, we investigate cycles in the object graph and simple structures called *leaf clusters*. We present some interesting properties of these structures, along with outlines of their respective proofs.

3.1 Object Graph Cycles

An object graph is a directed graph, where the direction of an edge reflects which object is referencing which. A cycle in an object graph indicates a codependency among objects, a system of objects that require structural connectivity in order to interoperate. A cycle in an object graph can only be formed by aggregations in our formalism since only aggregations are realized as object attributes. Cyclic (non-aggregation) associations reflect an interdependency of functionality, but not necessarily a structural dependency.

Whether a graph has cycles or not is an important factor when choosing an algorithm to draw it. Most graph drawing techniques assume a directed acyclic graph [3], and so cycles must be removed before these techniques are applied [4]. However, through analysis of the class diagram, we can determine whether a program can produce cycles or not. Specifically, we can prove the following property of class diagrams and object diagrams (assuming no type coercion):

Diagram Cyclicity Property*: A cycle can occur in the object diagram if and only if the class diagram is recursive.*

The proof of this property depends on the fact that edges in the object diagram are object references, and object references are the manifestation of aggregations in the class diagram. Furthermore, such a structural link between objects cannot occur unless there is a corresponding aggregation in the class diagram. The proof follows directly from these observations: for a cycle to exist in the object diagram, it must have corresponding cyclic aggregations; and if a recursive type is present in the class diagram, it can manifest as a cyclic object structure. As noted in Section 2.3, it is not necessary that a recursive type always yield cycles.

3.2 Leaf Clusters

The distribution of objects and structural links in object-oriented systems obeys a power law: there are very few objects with high degree, and there are very many objects with low degree [22]. The empirical evidence supports the intuition that, in an object-oriented system, a few objects act as mediators and coordinators while many more objects act as simple data containers. We wish to define more formally a methodology for detecting these simple data objects so that they can be rendered appropriately in object diagrams. Specifically, since these data objects are tightly coupled with the objects that reference them, we wish to draw leaf objects in close proximity with their dependents.

Let $type(v)$ be a predicate that returns the most specific runtime type of an object v. Then we can define a leaf class as follows:

Definition 6 *A **leaf class** in a class diagram is a class whose attributes, explicit and implied, are either primitive types or boxed types. We define a predicate $leafclass(C)$ that is true if C is a leaf class.*

Our definition of a *leaf class* describes those classes of a class diagram that have no outgoing associations. Implied attributes are those that are not listed in the attributes portion of the class but exist nonetheless, and they include attributes that are shown as labeled aggregations and those that are inherited from superclasses. *Boxed types* are those immutable classes that provide object wrappers around primitive value types. The property of being a leaf class is not inherited, since subclasses of leaf classes can define attributes that invalidate the requirements of being a leaf class; however, a class' superclasses must be leaf classes in order for it to be one. In terms of the object graph, we can assert that the following property holds:

Leaf Degree Property*: The objects of a leaf class will be leaves in the object graph.*

The proof of this property follows directly from our definitions. A leaf class has no outward aggregations by definition, including inherited ones. If an instance of a leaf class had an outgoing edge, then its class would have to have an outgoing aggregation, but this is a contradiction.

Definition 7 *Given an object graph $G = (V, E)$, a **leaf cluster** is a subgraph $C = (V' \subseteq V, E' \subseteq E)$ such that the following four properties hold:*

1. $V' = \{v*, v_1, v_2, \ldots, v_n\}$, *where* $n > 0$.
2. $leafclass(type(v_i))$ *for* $1 \leq i \leq n$.
3. $E' = \{(v*, v_i) \mid 1 \leq i \leq n \wedge (v*, v_i) \in E\}$.
4. $degree(v_i) = 1$ *for all* v_i *where* $1 \leq i \leq n$.

We call $v*$ the *leaf aggregator* and all other $v_i \in V'$ *leaf objects*. According to the definition, a leaf cluster is a two-level rooted tree in an object graph where all of the leaves are instances of a leaf class. Figure 1(b,d-e) shows a classic example of leaf clusters, where the `Data` class is the leaf class and `BST` is a leaf aggregator. The presence of compositions simplifies the detection of leaf clusters. Indeed, the very idea of a leaf cluster is closely related to the definition of composition: that an complex aggregate is made up simpler parts.

According to our definition, not all two-level rooted trees in the object graph are leaf clusters. This is necessary since program execution is dynamic, and the edges and vertices of the object graph can change with every execution step. An object that will eventually reference hundreds of other objects must at some point be created and initialized. At this point, we do not want to treat the object as if it were a leaf object, since we know from analysis of the class diagram that this is not the proper semantics of the object.

Let V be a set of classes and E be a set of directed relationships such that $G = (V, E)$ represents the structure of a class diagram. Then we can prove the following property of leaf clusters:

Leaf Prediction Property*: Leaf cluster classes can be identified in $O(|V| + |E|)$ time.*

In order to predict which classes may be come leaf clusters, we first build a partial order of the vertices based on reversed generalizations. That is, we visit the most general class first and the most specific class last. This topological sorting can be done via a breadth-first search in $O(|V| + |E|)$ time. The set of classes can be divided into two distinct subsets: the set of leaf classes L and the set of non-leaf classes $\bar{L}$. This can be done in a single pass through the set of classes, as long as they are processed in order of decreasing generality. Each class and relationship will be analyzed at most once, checking if superclasses are in L or $\bar{L}$, giving time complexity of $O(|V|+|E|)$. Finally, those classes that aggregate leaf classes can be identified by searcing backwards from part to whole along aggregations; this takes $O(|V|+|E|)$ time since in the worst case, all the classes may be leaf classes, and all aggregations would have to be followed.

3.3 Recursive Clusters

We showed in Section 2.3 that recursive types engender certain types of object structures. Within the context of that discussion, we looked at isolated recursive clusters; we did not consider the general case of structures that aggregate other clusters as well. In the worst case, a class diagram contains many aggregations and the resulting object diagram is a general graph. However, we can identify sets of classes that form *recursive clusters.*

Definition 8 *Given an object graph $G = (V, E)$, a* ***recursive cluster*** *is a subgraph $C = (V' \subseteq V, E' \subseteq E)$ such that the following four properties hold:*

- *C is connected.*
- *for all vertices in V', either their types are participants in a single recursive type τ or they are leaf classes.*
- *for all edges $(u, v) \in E'$, either $type(v) \in \tau$ or $leafclass(type(v))$.*
- *there is at most one vertex $v \in V'$ such that there exist edges $(u, v) \in E$ where $u \notin V'$.*

A recursive cluster is a subgraph of the object graph that is isolated, connected, and cohesive. It comprises a recursive object structure and the leaf objects aggregated by it. Furthermore, it has at most one entry point through which it is connected to the rest of the object graph. We make no other assertions about the structure of the recursive cluster. For example, we cannot assert whether recursive clusters are cyclic or not, although we can apply the technique from Section 3.1 to determine if it can possibly have cycles or not.

The key to predicting recursive clusters in the object diagram is the identification of recursive types in the class diagram. Let V be a set of classes and E be a set of directed relationships such that $G = (V, E)$ represents the structure of a class diagram. Then we can prove the following property of recursive types:

Recursive Type Prediction Property*: The classes in a recursive type can be identified in $O(|V \times E|)$ time.*

Since recursive types are defined by generalized aggregation paths, proving this property requires us to have an algorithm for detecting generalized aggregation paths. Given a class diagram, a starting class c, and an ending class d, there is a generalized aggregation path between c and d if there is a path of aggregations and generalizations as given in Definition 4. This can be determined by a standard graph searching algorithm such as breadth-first. The execution of this search is constrained by the number of relationships, giving a running time of $O(|E|)$. A recursive type is a type that has a generalized aggregation path to itself; applying the technique just presented, this takes $O(|E|)$ time. Testing all classes in the class diagram therefore takes $O(|V \times E|)$ time. In practice, it is useful for this algorithm to return a set of recursive types, each of which is expressed as a list of class identifiers.

4 Example

Once the class diagram and object graph are analyzed, the object diagram must be constructed. We differentiate here between the *object graph*, which is a mathematical abstraction, and the *object diagram*, which is a drawing of the object graph. There can be more than one object diagram for any object graph. The methodology we present in this section outlines the approach we have taken.

The overall strategy for layout involves the formation of *clusters* of vertices and the arrangement of these clusters [25]. We use the two types of clusters defined in Section 3: leaf clusters and recursive clusters. The clusters themselves can then be treated as vertices with non-trivial size, and they can be arranged using existing techniques.

In this section, we demonstrate how our approach works through an example. Figure 8 presents a UML class diagram for a simple expression parser. The expressions are list-based in the Lisp tradition, where each expression is a `List`, `EmptyList`, or `Atom`. `List` objects have a `head` and a `tail`, both of which are themselves expressions. The `Parser` takes a string as input and parses it into an expression with the help of a `Scanner`.

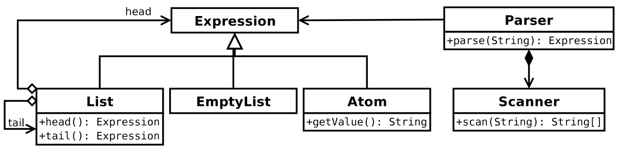

Figure 8: An architecture for a simple Lisp-style expression parser.

4.1 Cluster Formation

The static analysis of the class diagram requires two steps: identifying the leaf classes and finding recursive types. This can be done using the techniques described in Section 3. In our example, there are three leaf classes: `Scanner`, `EmptyList`, and `Atom`. Each of these classes has no outgoing aggregations, nor do they inherit any from their superclasses. There is one recursive type in Figure 8, the set {`Expression`, `List`}. `List` aggregates itself, making it a simple recursive type.

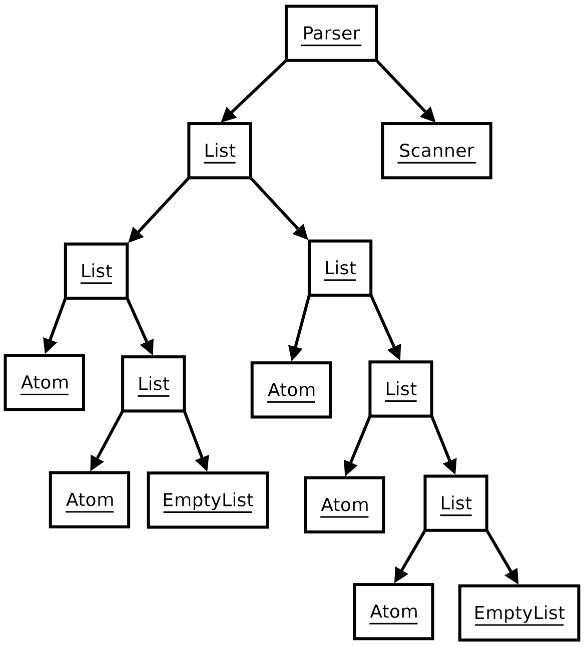

Figure 9: An object diagram for the Parser example. This drawing was produced using standard layered graph drawing techniques.

Figure 9 shows an object graph representing a state in the parser. This drawing was generated using a top-down layering followed by barycenter crossing-reduction and horizontal coordinate assignment step [4, 26]. At the visualized state of the program, the parser has built an expression tree for input of the form `((A B) C D)`, where `A`, `B`, `C`, and `D` are atoms. As predicted, the graph verifies that the instances of leaf classes are leaves in the graph. A recursive object structure has indeed arisen from `List` and `Expression`. We are not drawing all of the objects on the heap, only those objects whose classes are shown in the class diagram. In a language such as Java, it is nigh-impossible to draw the complete heap since even the simplest program has thousands of objects in memory at runtime.

4.2 Cluster Arrangement

Figure 10 shows a final drawing of the object graph where the clusters are laid out using an orthogonal drawing technique. Each leaf cluster is drawn using the same orthogonal technique, and the simple structural cluster is drawn using layered drawing techniques, as in the original drawing of Figure 9. The clusters have been aligned to enforce a straight-line drawing. In an implementation, a more general approach involving edge routing between clusters may be required.

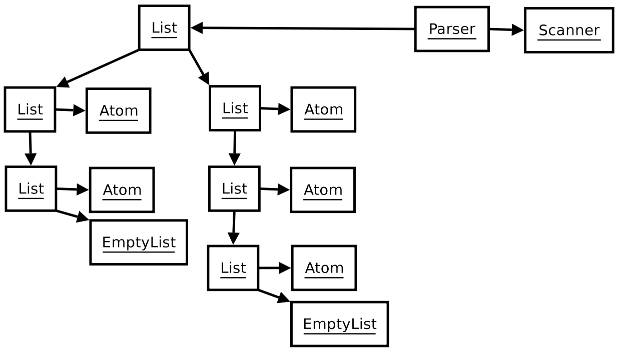

Figure 10: A drawing of the expression parser object graph using our techniques of class diagram and graph analysis

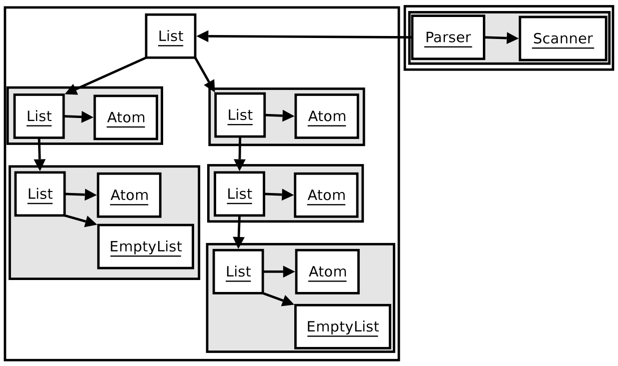

Figure 11: An alternate view of the drawing in Figure 10. In this view, leaf clusters are drawn in grey boxes and simple structural clusters are drawn in dotted rectangles.

Figure 11 provides an alternate view of the drawing where the leaf clusters are shown in grey boxes and the simple structural cluster is shown in a dotted rectangle. This view highlights how the different drawing techniques are used in each cluster and how these combine to form a good drawing.

It is important to notice that the drawings of Figures 10 and 11, are not optimal with respect to overall drawing size. The standard layered drawing technique shown in Figure 9 produces drawings that are compact in area. Our technique weakens this constraint in order to achieve more visible distinction between components. In Figure 10, we can see a clear separation between the expression tree, which is a simple recursive type, and the parser/scanner component.

The essence of our technique is in the declarative formation of clusters in the object graph. In practice, these clusters can be drawn many different ways. The leaf clusters of Figure 10 are drawn using a simple algorithm where the leaf aggregator is drawn on the left and leaf objects are drawn to the right of the aggregator. This technique is sufficient for our example, but a more robust solution would involve more advanced graph drawing techniques such as radial drawing [4], 2.5-dimensional drawings, or focus-directed hierarchical drawing [2].

Before drawing recursive clusters, we first remove its leaf clusters and replace them with vertices of nontrivial area. This technique follows from Definition 8, in which we allow leaf clusters to be contained with recursive clusters. We have drawn the recursive cluster using a hierarchical technique [26]. This has produced a good drawing since the recursive cluster is itself a tree. In general, this drawing step would require converting the graph into a directed acyclic graph first. Other techniques can be used to arrange clusters as well, including blob layouts [15] or other orthogonal methods [4].

5 Conclusions and Future Work

This paper presents some useful strategies as shown in the example of Section 4. User responses indicate that these aesthetic properties enhance program comprehension, but making a scientific claim on this front requires extensive user testing. We plan to incorporate this work into JIVE, a tool for advanced visualization of program execution [10], which will enable a more formal analysis of the effectiveness of this strategy.

Class diagrams are static, but object graphs are dynamic. We have treated object graphs as isolated artifacts, but in practice, a series of object graphs is generated as the program executes. In an system that draws these graphs dynamically, it is important that drawings be presented in such a way that the user maintains his or her mental map of the meaning of each component [5]. An important area of future research involves investigating how program-specific properties can be incorporated into techniques of dynamic graph drawing [21].

Our technique is generally applicable to object-oriented languages, but we are interested specifically in how these techniques apply to Java since it is the language visualized in JIVE [10]. The recently-released Java 5.0 incorporates syntactic tools to ease the use of the Collections API through its support for generics[28]. We expect the number of programs using the Collections library to increase, especially in educational environments [29]. It is therefore advantageous to have custom techniques to draw these collections in a way that mirrors their semantics. We are currently experimenting with algorithms to draw such collection classes as `LinkedList`, `HashMap`, and `Stack`. This is an instance of the general problem of drawing object diagrams in a way that reflects their semantics, but it is clear that static analysis (that is, knowing the specific collection class and its content type) will assist in the drawing.

As mentioned in the introduction, JIVE uses object diagrams to represent state and sequence diagrams to represent execution history. Both types of diagrams are generated dynamically as a program executes [10]. It is possible that this same form of class diagram analysis may assist the dynamic drawing of sequence diagrams. One of the biggest challenges in sequence diagram drawing is that object lifelines must be drawn sequentially, and so the amount of space needed is linearly proportional to the number of interacting objects [1]. Focus-dependent techniques could be used to produce multiple, modular views of the sequence diagram [2]. It seems that the leaf clusters and recursive cluster models would be useful for defining object clusters in the sequence diagram, but more study and experimentation is necessary before any qualititive claims can be made.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison–Wesley, Reading, MA, USA, 1999.

[2] F. Boutin and M. Hascoët. Focus dependent multi-level graph clustering. In *Proceedings of*

the working conference on Advanced visual interfaces, pages 167–170. ACM Press, 2004.

[3] J. Díaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Comput. Surv.*, 34(3):313–356, 2002.

[4] P. Eades, G. Di Battista, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualisation of Graphs*. Prentice-Hall, New Jersey, 1999.

[5] P. Eades, W. Lai, K. Misue, and K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of Compugraphics 91*, pages 24–33, 1991.

[6] M. Eiglsperger, M. Kaufmann, and M. Siebenhaller. A topology-shape-metrics approach for the automatic layout of uml class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 189–ff. ACM Press, 2003.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1995.

[8] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Trans. Softw. Eng.*, 19(3):214–230, 1993.

[9] P. Gestwicki and B. Jayaraman. Interactive visualization of Java programs. In *IEEE Symposium on Human-Centric Computing, Languages, and Environments*, pages 226–235, September 2002.

[10] P. Gestwicki and B. Jayaraman. Jive: Java interactive visualization environment. In *OOPSLA 2004 Conference Companion*, pages 615–616, 2004.

[11] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[13] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: putting icing on the uml cake. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 301–314. ACM Press, 2004.

[14] C. Gutwenger, M. Jünger, K. Klein, J. Kupke, S. Leipert, and P. Mutzel. A new approach for visualizing UML class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 179–188. ACM Press, 2003.

[15] D. Harel and G. Yashchin. An algorithm for blob hierarchy layout. In *Proceedings of the working conference on Advanced visual interfaces*, pages 29–40. ACM Press, 2000.

[16] W. Harrison, C. Barton, and M. Raghavachari. Mapping uml designs to java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 178–187. ACM Press, 2000.

[17] T. D. Hendrix, I. James H. Cross, and L. A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 387–391. ACM Press, 2004.

[18] T. Jacobs and B. Musial. Interactive visual debugging with uml. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 115–122. ACM Press, 2003.

[19] B. Jayaraman and C. Baltus. Visualizing program execution. *Proc. IEEE Symp. on Visual Languages*, pages 30–37, September 1996.

[20] R. Kollmann and M. Gogolla. Application of UML associations and their adornments in design recovery. In *Working Conference on Reverse Engineering*, pages 81–91, 2001.

[21] S. C. North. Incremental layout in dynadag. In *Graph Drawing*, pages 409–418, 1995.

[22] A. Potanin, J. Noble, M. Frean, and R. Biddle. Scale-free geometry in object-oriented programs. *Communications of the ACM*. (to be published).

[23] H. C. Purchase, M. McGill, L. Colpoys, and D. Carrington. Graph drawing aesthetics and the comprehension of uml class diagrams: an empirical study. In *Australian symposium on Information visualisation*, pages 129–137. Australian Computer Society, Inc., 2001.

[24] S. P. Reiss. An overview of bloom. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–5. ACM Press, 2001.

[25] T. Roxborough and A. Sen. Graph clustering using multiway ratio cut. In G. Di Battista, editor, *Proc. 5th Int. Symp. Graph Drawing, GD*, number 1353, pages 291–296. Springer-Verlag, 18–20 1997.

[26] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.*, SMC-11(2):109–125, 1981.

[27] Sun Microsystems. *Java 2 Platform Standard Edition 5.0 API Documentation*, 2004.

[28] Sun Microsystems, Inc. *Java 2 SDK, Standard Edition, Version 1.5.0 Summary of New Features and Enhancements*, 2004. In the Java 2 SDK 1.5 documentation.

[29] P. Ventura, C. Egert, and A. Decker. Ancestor worship in cs1: on the primacy of arrays. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 68–72. ACM Press, 2004.