

Dynamically Composable Collaborations with Delegation Layers

Klaus Ostermann

Siemens AG, CT SE 2, D-81730 Munich, Germany
Klaus.Ostermann@mchp.siemens.de

Abstract. It has been recognized in several works that a slice of behavior affecting a set of collaborating classes is a better unit of reuse than a single class. Different techniques and language extensions have been suggested to express such slices in programming languages. We propose delegation layers, an approach that scales the OO mechanisms for single objects, such as delegation, late binding, and subtype polymorphism, to sets of collaborating objects. Technically, delegation layers combine and generalize delegation and virtual class concepts. Due to their runtime semantics, delegation layers are more flexible than previous compile time approaches like mixin layers.

1 Introduction

In the early days of object-oriented programming there has been a general agreement that the class should be the primary unit of organization and reuse. However, over the years it has been recognized that a slice of behavior affecting a set of collaborating classes is a better unit of organization than a single class. In the face of these insights, mainstream programming languages have been equipped with lightweight linguistic means to group sets of related classes, for example name spaces in C++ [11] or packages and nested classes in Java [1]. On the other hand, the research community has developed a great deal of models related to *collaboration-* or *role-model based design*, for example [4, 15, 29, 34].

Our point of view is that we should not try to invent a completely new kind of module for grouping classes just to realize (sooner or later) that we need means to express variants, hide details, have polymorphism etc. Instead, we propose a model within which all the concepts that proved so useful for *single* classes/objects, for example inheritance, delegation, late binding, and subtype polymorphism, automatically apply to *sets* of collaborating classes and objects.

In particular, we deal with the question of how sets of collaborating classes can be defined and composed in terms of different variants (*layers*) of a base collaboration. The running example in this paper is a graph collaboration with classes like `Node` and `Edge` and variations of this collaboration for colored graphs and weighted graphs.

One of the most advanced approaches with respect to our goals is the *mixin layer* approach by Smaragdakis and Batory [31]. Mixin layers allow (a) sets of

classes (which represent a particular collaboration and are implemented as nested classes of an outer class) to inherit from other sets of classes, and (b) the composition of different variants of a base collaboration. With regard to our running example, this means we can (a) implement a `Graph` collaboration with `Node` and `Edge` classes and refine the `Graph` collaboration to `ColoredGraph` or `WeightedGraph` via inheritance, and (b) combine `ColoredGraph` and `WeightedGraph` to a `ColoredWeightedGraph`.

Technically, the most important difference between our *delegation layer* approach and mixin layers is that the mixin layer notion of multi-class mixin-inheritance is replaced by multi-object delegation¹. For those readers who have never heard of mixin-inheritance [5] or delegation [20] (we will elaborate on that in the paper), this can be tentatively summarized as: With mixin layers, everything happens on classes at compile time, whereas with delegation layers, everything happens on objects at runtime.

This has a deep impact on the semantics and expressiveness of the model. In particular, delegation layers have the following two properties:

- **Polymorphic runtime composition:** In our approach, a collaboration is composed at runtime by combining different delegation layers. Since delegation layers are subject to subtype polymorphism, the code which combines the layers is decoupled from the specific layers to be composed. For example, we may combine a `ColoredGraph` with a `Graph g`, but at runtime, `g` may actually refer to an instance of `WeightedGraph`.
- **Local on-the-fly extensibility:** We can extend a group of collaborating objects' behavior on-the-fly, whereby these behavior extensions are local, meaning that after the extension both the original and modified behavior of the object group are simultaneously accessible. For example, we may have an existing graph instance `g` with a set of node and edge objects and extend `g` with all its nodes and edges to be a colored graph `cg`. After the extension, the nodes and edges of the graph behave as a colored graph if they are accessed via `cg` and as a usual graph if they are accessed via `g`. We may even have multiple independent color extensions of a specific graph denoting different colorings of the same graph.

These properties are consequences of the runtime semantics of delegation layers. In addition, our approach eliminates two subtle flaws of the mixin layer approach related to polymorphism and consistency:

- **Polymorphism:** We define a notion of subtyping among collaborations which guarantees substitutability and allows us to use a compound collaboration where an instance of a particular layer is expected if and only if this layer is a part of the compound collaboration. For example, a graph that is both colored and weighted can be used where a colored graph is expected.

¹ Please note that in contrast to the frequent use of the term delegation as a synonym for forwarding semantics, in this paper it stands for dynamic, object-based inheritance as defined in [20].

Thus the advantages of standard OO subtyping (reusability, decoupling etc.) are transferred to collaboration inheritance. In general, this property does not apply for mixin layers.

- **Composition consistency:** Our approach guarantees that all operations inside a compound collaboration are applied to the composite collaboration rather than to a specific layer alone. In particular, this proposition holds for constructor calls, thereby eliminating a composition anomaly of mixin layers.

Composing and extending collaborations at runtime yields type safety and consistency questions not emerging with compile time composition. In order to give answers to these questions, our model combines delegation techniques with virtual classes [21], family polymorphism [12], and a wrapper technique that is based on the idea of lifting and lowering as described in [26]. Although - to the best knowledge of the author - delegation has never been combined with virtual classes before, the interplay between these two mechanisms is elegant and natural.

The rest of this paper is structured as follows: Sec. 2 elucidates the concept of composable collaborations and gives a short overview of mixin layers. In addition, it emphasizes the weaknesses of mixin layers with respect to the aforementioned benefits we aim at. Sec. 3 and 4 introduce simple variants of delegation and virtual classes with family polymorphism as extensions of Java [1]. Sec. 5 shows how delegation and virtual classes interact and introduces the notion of delegation layers. Sec. 6 elaborates on on-the-fly extensions and the impact of delegation layers on sharing and aliasing. Sec. 7 discusses related work. Sec. 8 summarizes and indicates areas of future work.

2 Collaboration Composition and Mixin Layers

The rationale behind collaboration composition is that sets of collaborating classes can be defined and composed in terms of different variants (*layers*) of a base collaboration. Consider the situation in Fig. 1. It shows two collaborations `ColoredGraph` and `WeightedGraph` that inherit from a base collaboration `Graph`. The `Graph` collaboration defines classes `Node`, `Edge` and `UEdge`. The graphs in this example are assumed to be directed in general, and the class `UEdge` represents undirected edges which enter themselves in the adjacency list of *both* nodes. The subcollaborations `ColoredGraph` and `WeightedGraph` extend the base collaborations' classes. For example, the class `ColoredGraph.Edge` extends `Graph.Edge` by an additional association to `Color`. The class `WeightedGraph.Edge` adds a field `float weight` and `WeightedGraph.Node` overrides the inherited `shortestPath()` method in order to consider the edge weights.

The key issue is the ability to compose different variants of a base collaboration. For example, we may want to create a graph that is both colored and weighted by means of the collaborations in Fig. 1. Fig. 2 demonstrates the desired semantics of a combination `WeightedGraph(ColoredGraph(Graph))`: The

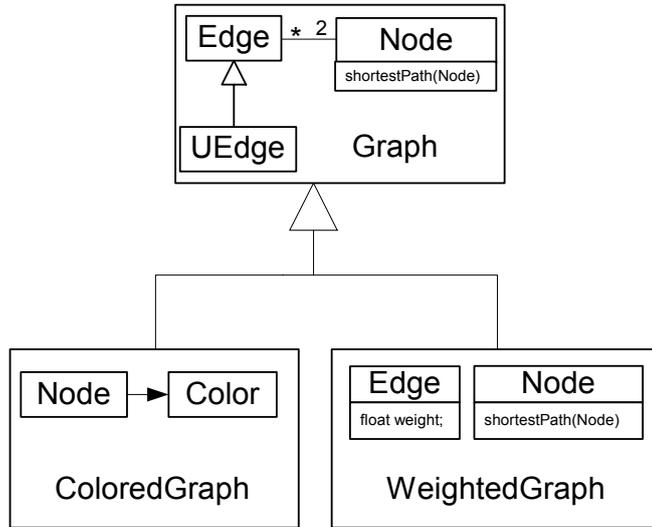


Fig. 1. Collaboration inheritance

collaborations are organized in layers according to the order in the composition expression; i.e. the outermost `WeightedGraph` collaboration is at the bottom, in the middle the `ColoredGraph`, and the `Graph` collaboration on top.

All inner classes are organized according to the definition of the outer abstractions. For example, in the context of the compound collaboration in Fig. 2, `WeightedGraph.Edge` is a subclass of `ColoredGraph.Edge`, and the compound `UEdge` is a subclass of the *compound* `Edge` class rather than of `Graph.Edge`.

In other words, superclasses of the collaboration classes are replaced by subclasses of the annotated superclass. This kind of class combination is commonly known as *mixin-inheritance* [5]. Mixin-inheritance relaxes the strong coupling between a class and its superclass by enabling the instantiation of a class with different superclasses. This property renders mixin-inheritance suitable for defining and combining uniform incremental extensions of a class.

Mixin layers [31] scale this concept to multi-class granularity. In [31], the authors propose the usage of C++ [11] to implement mixin layers. Fig. 3 shows C++ mixin layers corresponding to Fig. 1 and 2. The basic technique is that the collaborating classes are implemented as nested classes of an outer class representing the collaboration. Subcollaborations are implemented as template classes with a parameterizable superclass. This superclass also determines the superclasses of the nested classes. Since C++ does not support F-bounded polymorphism [8], the template parameters are not explicitly bounded, but they should be thought of as being restricted to subclasses of `Graph`. The `typedef` statements at the bottom of Fig. 3 compose different collaboration variants. The type `CG` designates a colored graph, and `CWG` designates a colored weighted graph.

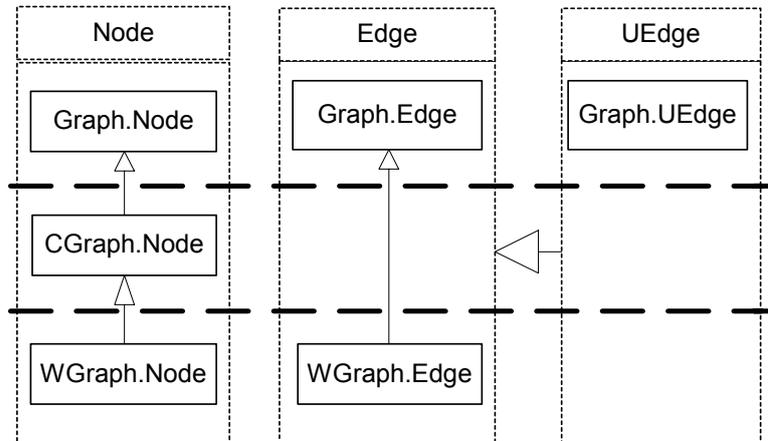


Fig. 2. Layer combination: `WeightedGraph(ColoredGraph(Graph))`:

In the following, we want to elaborate on the two flaws of mixin layers mentioned in the introduction.

- **Polymorphism:** Mixin layers have two flaws concerning polymorphism. First, subtyping among collaborations is too restrictive. Consider for example the two types `CWG` and `CG` in Fig. 3. Although a colored weighted graph of type `CWG` has all features of a colored graph type (`CG`), the former one is no subtype of the latter one². Secondly, in the cases where subtyping is possible, the effect of substitution is not as expected. Consider the example in Fig. 4. If the method `createTransitiveHull` is called with an instance of a colored graph `CG`, the `new` statements will still create instances of `Graph.Node` and `Graph.Edge` rather than of their corresponding implementations for colored graphs. The problem is that the constructors in a `new` statement are statically bound to a specific implementation and we have no means to express that the `new` statements should instantiate the classes that are appropriate in the specific collaboration represented by `g`. Please note that the factory pattern [13] is in general no satisfactory solution because this pattern needs to anticipated and cannot be applied to superclasses (i.e., we cannot retrieve the superclass of a nested class via a factory object).
- **Composition consistency:** Inside a compound collaboration, all operations should be applied to the composite collaboration rather than to a specific layer. This is in general not true for constructor calls in mixin layers. Consider for example the compound `UEdge` class in Fig. 2. In a weighted graph, the weight property should of course also apply to `UEdge`. This means

² However, this flaw can be attributed to the C++ template implementation and is no conceptual weakness of mixin layers

```

class Graph {
public:
class Node {
public:
NodeList shortestPath(Node *t) {...}
};
class Edge { ... };
class UEdge: public Edge { ... };
};
template <class SuperGraph>
class ColoredGraph : public SuperGraph {
public:
class Node: public SuperGraph::Node {
Color color;
...
}
};
template <class SuperGraph>
class WeightedGraph : public SuperGraph {
class Edge: public SuperGraph::Edge {
float weight;
...
}
class Node: public SuperGraph::Node {
public:
NodeList shortestPath(Node *t) {...}
}
};
typedef ColoredGraph<WeightedGraph<Graph> > CWG;
typedef ColoredGraph<Graph> CG;

```

Fig. 3. Graph Example with C++ mixin layers

```

class Client {
void createTransitiveHull(Graph *g, Graph::Node *n) {
... Graph::Node *m = currentNode->neighbor(i); ...
if ( ! n->isNeighbor(m) ) {
Graph::Edge *e = new Graph::Edge(n,m); ...
}
}
};

```

Fig. 4. Restricted polymorphism in the mixin layer approach

that in the context of a weighted graph, `UEdge` should inherit from the *compound* `Edge` rather than from `Graph.Edge` (see also Fig. 2). The same argument also applies to `new` statements inside a collaboration. If the class to be created is a participant of the collaboration, we expect a corresponding `new` statement to create an instance of the respective *compound* participant class. However, in the mixin layer approach the constructor calls refer to a fixed implementation in both cases. For example, the `UEdge` class in Fig. 3 is always a subclass of `Graph.Edge` and not of `WeightedGraph.Edge`, even in the context of the weighted graph collaboration.

The second problem has also been acknowledged in [31]. We think that it can be seen as a variant of the *self problem* [20], a.k.a. *broken delegation* [14]: In a composite component, all actions should be applied to the composite component, rather than to an individual part of it. The original formulation of the self problem refers to method calls; in our case, it refers to constructor calls.

3 Delegation

This section introduces the first building block of delegation layers, namely a simple variant of delegation as an extension of Java. Delegation means that objects inherit from other objects, with roughly the same semantics as classes which inherit from other classes. An object `o` that inherits from (*delegates to*) another object `p` is called *child* of `p`, and `p` is a *parent* of `o`.

To make the discussion simple, we restrict ourselves to *static delegation*, meaning that the parent of an object can be set at runtime, but once the parent reference is initialized, it cannot be changed, similar to a `final` variable in Java. This restriction avoids many problems which are not in the scope of this paper; see [19, 27].

Consider the situation in Fig. 5. It shows classes `Graph`, `ColoredGraph` and `WeightedGraph` as well as some demonstration code that uses delegation. In our approach, we unify standard inheritance and delegation as follows: In a `new WeightedGraph()` expression for a class `WeightedGraph` as in Fig. 5, we may *optionally* specify a parent object (delimited by `<>`) that has to be a subtype of the original superclass `Graph`. For example, let `ColoredGraph` be another subclass of `Graph`. Then `new WeightedGraph()` creates an instance of `WeightedGraph` with superclass `Graph` (usual semantics), and `new WeightedGraph<cg>()` creates an instance of `WeightedGraph` with parent `cg` (see Fig. 5). In the latter case, the parent object replaces the superclass.

The unification of delegation and inheritance has two advantages. First, the usage of a class with a different superclass as initially intended does not have to be anticipated. Second, we have a *default* superclass/parent, so that it becomes easier to create instances of such a class. In the following, we will treat the direct instantiation of an object (without specifying a parent object) as an abbreviation for assigning an instance of the superclass as parent object. For example, `new`

`ColoredGraph()` is an abbreviation for `new ColoredGraph<new Graph>()`³. The parent object of an object `o` is always available via the implicit `super` field `o.super`.

```
class Graph {
    private String info = "SomeInfo";
    public String setInfo(String s) { info = s; }
    String toString() { return "Graph, info="+info; }
    void printInfo() { print(this.toString()); }
}
class ColoredGraph extends Graph {
    String toString() { return "Colored"+super.toString();}
}
class WeightedGraph extends Graph {
    String toString() { return "Weighted"+super.toString();}
}
// demo code
Graph g = new WeightedGraph();
g.printInfo(); // prints "WeightedGraph, info=SomeInfo"
Graph cg = new ColoredGraph();
cg.printInfo(); // prints "ColoredGraph, info=SomeInfo"
Graph wg = new WeightedGraph<cg>();
wg.printInfo(); // prints "WeightedColoredGraph, info=SomeInfo"
cg.setInfo("OtherInfo");
wg.printInfo(); // prints "WeightedColoredGraph, info=OtherInfo"
ColoredGraph cg2 =
    (ColoredGraph) wg; // succeeding cast due to transparency
```

Fig. 5. Code example for delegation

The key issue in combining classes and objects via delegation is the treatment of the `this` and `super` pseudo variables. This is illustrated in the demonstration code in Fig. 5 and in Fig. 6: The `this` pseudo variable refers to the *receiver* of a method call, and `super` refers to the (possibly dynamically assigned) parent/superclass. The implications are illustrated by the `printInfo()` calls in Fig. 5. It is important to understand that the “value” of `this` is not fixed but depends on the receiver of a message. For example, in the context of a method call to `cg`, all `this` pointers refer to the instance of `ColoredGraph` rather than to the `WeightedGraph` instance.

Delegation is more than just composing classes at runtime. An important property of delegation is that parent objects may be *shared*. In Fig. 5, both the `cg` instance variable and the parent reference of `wg` refer to the same object. This

³ For the sake of simplicity we assume that every class has only a single no-argument constructor.

is demonstrated by the `cg.setInfo()` call which affects `wg` due to the shared `ColoredGraph` object.

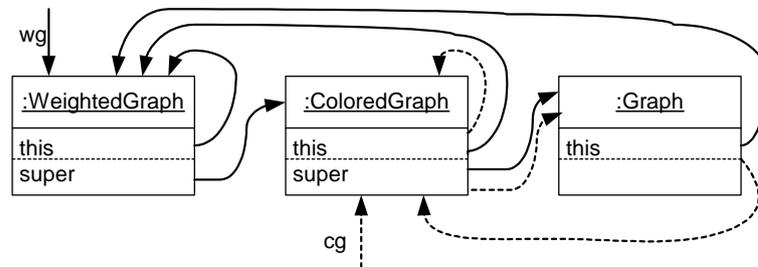


Fig. 6. Meaning of `this` and `super` in a delegation relationship as in Fig. 5. The non-dashed lines represent the behavior if the objects are accessed via `wg` and the dashed lines represent the behavior if the objects are accessed via `cg`.

A property of delegation that has been postulated by Büchi and Weck [3] is *transparency*, meaning that an object is a subtype of the *dynamic* type of its parent. This property is relatively straightforward in the context of static delegation⁴. In our example (Fig. 5), transparency means that the dynamic cast in the last line succeeds. We will see that incorporation of transparency supports the elimination of polymorphism problem indicated in Sec. 2 (“better support for polymorphism”).

For further details about the integration of delegation into a statically typed language, we refer to the existing approaches, e.g., [18, 3, 27].

4 Virtual Classes

Virtual classes are the second important building block for delegation layers. Virtual classes are a concept from the Beta programming language [21, 22] (in Beta known as *virtual pattern*). The basic idea is that the notions of overriding and late binding should also apply to nested classes, similarly to overriding and late binding of methods.

Consider the class `Graph` in Fig. 7. The nested classes `Node`, `Edge` and `UEdge` are declared as *virtual classes* with a `virtual` modifier (corresponds to `:<` in Beta), meaning that these classes can be overridden by subclasses of the enclosing class. In contrast to methods in Java, nested classes are not virtual by default, because a virtual class has important typing implications.

⁴ In models that support full dynamic delegation, transparency implies serious typing problems.

```

class Graph {
    virtual class Node {
        void foo() { Edge e = new Edge(); }
        NodeList shortestPath(Node n) { ... }
    }
    virtual class Edge { ... }
    virtual class UEdge extends Edge { ... }
}

class ColoredGraph extends Graph {
    override class Node { Color color; ... }
}

class WeightedGraph extends Graph {
    override class Node {
        NodeList shortestPath(Node n) {...}
    }
    override class Edge { float weight; ... }
}

```

Fig. 7. Virtual classes

The classes `ColoredGraph` and `WeightedGraph` in Fig. 7 override virtual classes of their superclass. The meaning of an `override` declaration (corresponds to `::<` in Beta) such as the `override class Node` declaration in `ColoredGraph` is that - in the context of `ColoredGraph` - the class `Graph.Node` is replaced by the class `ColoredGraph.Node`. The latter one is automatically a subclass of the former one, and hence has all methods and fields of `Graph.Node` plus an additional `color` field. Since an overriding virtual class is automatically a subclass of the overridden class, the single inheritance link is already allocated, such that an overriding class cannot extend another class.

As mentioned before, the rationale behind virtual classes is that overriding and late binding should uniformly apply to methods as well as virtual classes. Late binding of *methods* means that the receiver *object* determines the method implementation to be executed. If this principle is applied to virtual classes, it becomes clear that virtual classes should also be properties of *objects* of the enclosing class, rather than properties of the enclosing class itself. This is in the vein of the family polymorphism approach [12].

Virtual classes being properties of an object means that all references to a virtual class are resolved via an instance of the enclosing class. In our approach, we apply the Java scoping rules for method calls to virtual classes as well, meaning that all references to a virtual class are implicitly resolved via the corresponding `this`. Fig. 8 makes the implicit scoping of Fig. 7 explicit. For example, the type declaration `Edge e` in `foo()` is a shorthand for `Graph.this.Edge` (the notation `Graph.this` refers to the instance of the enclosing class). Similarly, `UEdge` in

Graph is a subclass of `this.Edge` (rather than of `Graph.Edge`) and `ColoredGraph.Node` extends `super.Node` (rather than `Graph.Node`).

```
class Graph {
  virtual class Node {
    void foo() { Graph.this.Edge e = new Graph.this.Edge(); }
    NodeList shortestPath(Graph.this.Node n) { ... }
  }
  virtual class Edge { ... }
  virtual class UEdge extends this.Edge { ... }
}

class ColoredGraph extends Graph {
  override class Node extends super.Node { Color color; ... }
}

class WeightedGraph extends Graph {
  override class Node extends super.Node {
    NodeList shortestPath(WeightedGraph.this.Node n) {...}
  }
  override class Edge extends super.Edge { float weight; ... }
}
```

Fig. 8. Virtual classes are properties of *objects* of the enclosing class.

Consequently, `Graph.Node` is no longer a valid type annotation: The treatment of virtual classes as properties of objects stretches out to the client code of a class as well. For example, the family polymorphism version of Fig. 4 is shown in Fig. 9: Type annotations and constructor calls for virtual classes are all redirected via an instance of the enclosing class.

For type checking reasons, variables that are used inside type declarations have to be `final`. Otherwise, the type `g.Node` of a variable could change due to an update of `g`.

```
class Client {
  void createTransitiveHull(final Graph graph, graph.Node n) {
    ... graph.Node m = currentNode.neighbor(i); ...
    if ( ! n.isNeighbor(m) ) {
      graph.Edge e = new graph.Edge(n,m); ...
    }
  }
}
```

Fig. 9. Family polymorphism version of Fig. 4

In contrast to Fig. 4, `createTransitiveHull` in Fig. 9 works with arbitrary subclasses of `graph` and without compromising type safety. Similarly, a statement like `node1.shortestPath(node2)` can be statically proved type-safe, if `node1` and `node2` are both of type `g.Node`, although at runtime `g` may refer to an instance of an arbitrary *subclass* of `Graph`. This demonstrates that the treatment of virtual classes as properties of objects is an alternative to other approaches for retaining type safety in the presence of virtual classes, such as final bindings [33] or type-exact variables [7].

For more details about virtual classes with family polymorphism, we refer to [12].

5 Delegation Layers

Delegation layers are the result of combining delegation with virtual classes. In the following, we want to elaborate on the interplay between these two mechanisms.

Reconsider the semantics of our virtual class mechanism as demonstrated in Fig. 7 and 8. All references to virtual classes are actually resolved via the implicit `this` and `super` pseudo variables of the enclosing object.

Our delegation mechanism implies that the “meaning” of `this` and `super` can be altered at runtime. Consider the second `new` expression in Fig. 10. It creates an instance of `WeightedGraph` and assigns an instance of `ColoredGraph` as parent of the weighted graph object. The meaning of `this` and `super` in the context of `g` has already been illustrated in Fig. 6.

The crucial point is that virtual classes and delegation, if combined, interact due to their influence/dependency on the semantics of `this` and `super`. The semantics of this interaction, which is illustrated in Fig. 11, can be derived from Fig. 6 and 8. Consider, for example, the superclass declaration “`extends super.Node`” in `WeightedGraph.Node` (Fig. 8). In the context of `g`, `super` refers to an instance of `ColoredGraph`, therefore the parent of `WeightedGraph.Node` is an instance of `ColoredGraph.Node`. Similarly, the superclass declaration “`extends this.Edge`” in `Graph.UEdge` binds the parent of `UEdge` to an instance of `WeightedGraph.Edge`. The boxes with bold frame in Fig. 11 represent the composite classes `g.Node`, `g.Edge`, and `g.UEdge`.

```
main() {
    Graph cg = new ColoredGraph();
    Graph g = new WeightedGraph<cg>();
    g.Node n = ...;
    new Client().createTransitiveHull(g,n);
    ColoredGraph cg2 = (ColoredGraph) g; // succeeding dynamic cast
}
```

Fig. 10. Delegation layers

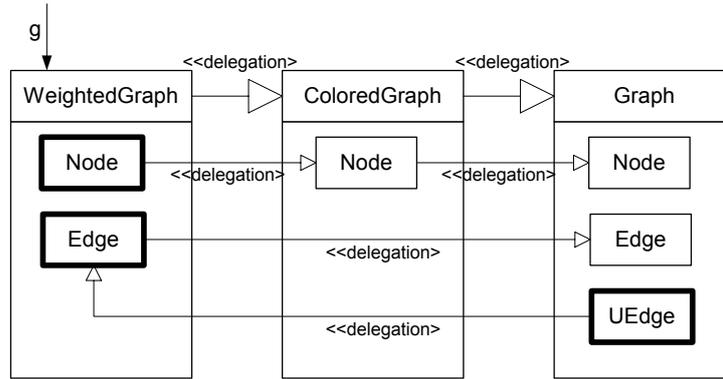


Fig. 11. Recursive delegation

Generally speaking, the combination of virtual classes and delegation effects the delegation relationship to spread to the nested virtual classes of the enclosing object. This is exactly the semantics that is required to obtain the composition behavior indicated in Fig. 2; cf. Fig. 2 and Fig. 11.

Let us revisit the delegation layer approach with respect to the goals stated in the introduction (except on-the-fly extensibility, which is the subject of the next section).

- **Polymorphic runtime composition:** Due to the use of delegation, the composition happens at runtime, and the composition code does not need to know the exact classes of the layers. Note, for example, that the static type of the parent reference `cg` in Fig. 10 is `Graph`, although it actually refers to an instance of `ColoredGraph`, which might have been passed as a method argument as well.
- **Polymorphism:** In Sec. 2 we stated two different shortcomings related to polymorphism with mixin layers. The first one is the restricted subtyping. For example, a colored weighted graph `CWG` is no subtype of a colored graph `CG` in Fig. 3. Compare this with the last line in Fig. 10. Due to the introduction of transparency (see Sec. 3), an instance of a compound collaboration can be dynamically casted to the type of all participants. In general, if $C = C_1(C_2(\dots(C_n)\dots))$ is a composed collaboration, C is a subtype of C_i for $i = 1, \dots, n$ in the delegation layer approach. With mixin layers, on the other hand, C is only a subtype of $C_i(C_{i+1}(\dots(C_n)\dots))$ for $i = 1, \dots, n$.
The second flaw is that the effect of subsumption is not as expected, exemplified by Fig. 4. This problem ceases to exist in our approach, because the class which is instantiated in response to a constructor call is determined at runtime, depending on the instance of the enclosing class (see Fig. 9).
- **Composition consistency:** We postulated that inside a compound collaboration, all operations should be applied to the composite collaboration rather than to a specific layer. With mixin layers, this property is violated, cf. the

discussion in Sec. 2. With delegation layers, the class `UEdge` is a subclass of the *compound* `Edge` class (see Fig. 11), and the `new Edge()` statement in `foo()` creates an instance of the *compound* `Edge` (cf. Fig. 7 and 8).

6 Hot State and On-the-fly Extensions

So far, we have avoided to introduce state into the classes in our examples. At first sight, it seems as if this implies both a semantic and a typing problem. Consider the code in Fig. 12. The `Node n` instance variable is initialized to an instance of `Graph.Node` (and not `ColoredGraph.Node`). However, if this graph instance was extended by an instance of `ColoredGraph`, the same (identical) node would suddenly have to be a colored node.

```
class Graph {
    Node n;
    Node getNode() { return n; }
    void setNode(Node n) { this.n = n; }
    ... // as in Fig. 7
}
// demo code
final Graph g = new Graph();
g.setNode(new g.Node());
g.Node node = g.getNode(); // OK
final ColoredGraph cg = new ColoredGraph<g>();
cg.Node cnode = cg.getNode(); // Type Error ?
cnode.color = Color.RED; // ??
```

Fig. 12. Potential problems due to hot state

The cause of this problem is that the type of the instance variable `Node n` is non-constant. Recall that `Node n` is an abbreviation for `this.Node n` (see Sec. 4). Hence `this` refers to an instance of `Graph` if `n` is accessed via `g`, and refers to an instance of `ColoredGraph` if `n` is accessed via `cg`. We call such state whose type depends on the enclosing `this` *hot state*.

We found a mechanism that turns this problem into a feature. It is based on the idea of *lifting* and *lowering* as described in [26] but adapted to the specific needs of our model. The basic idea is that, in the context of a colored graph, a node `n` can be automatically lifted to a colored node by creating an instance of `ColoredGraph.Node` that delegates to `n`. In order to make this approach sound, it is essential that two subsequent liftings for the same node yield the same colored node.

Fig. 13 shows pseudo code indicating the operational semantics of the lifting and lowering operation. Please note that the programmer does not write this code: The code is just an illustration of the language semantics in terms of OO constructs.

Every class **C** maintains a map (hashtable) for every virtual class which is overridden in that class. For example, **ColoredGraph** has a map `nodeMap`, and **WeightedGraph** has the maps `nodeMap` and `edgeMap`. In addition, a class **C** has a lifting and a lowering operation `liftV(V v)` resp. `lowerV(V v)` for each virtual class **V** that is (a) defined or (b) overridden in **C**. In case (a), the lifting and lowering operations simply return their argument. In case (b), the lifting operation lifts an instance of the base virtual class (e.g., `Graph.Node`) to an instance of the overriding virtual class (`ColoredGraph.Node`), and the lowering operation lowers an instance of the overriding virtual class (e.g., `ColoredGraph.Node`) to an instance of the base virtual class (`Graph.Node`).

```

class Graph {
    // begin internal structure pseudocode
    Graph.Node liftNode(Graph.Node n) { return n; }
    Graph.Node lowerNode(Graph.Node n) { return n; }
    Graph.Edge liftEdge(Graph.Edge e) { return e; }
    Graph.Edge lowerEdge(Graph.Edge e) { return e; }
    // end internal structure pseudocode
}
class ColoredGraph extends Graph {
    // begin internal structure pseudocode
    private Map nodeMap = new HashMap();

    ColoredGraph.Node liftNode(Graph.Node n) {
        ColoredGraph.Node result = (ColoredGraph.Node) nodeMap.get(n);
        if (result == null) {
            result = new ColoredGraph.Node<super.liftNode(n)>();
            nodeMap.put(n,result);
        }
        return result;
    }
    Graph.Node lowerNode(ColoredGraph.Node n) {
        Graph.Node result = super.lowerNode(n.super);
        nodeMap.put(result, n);
        return result;
    }
    // end internal structure pseudocode
}

```

Fig. 13. Lifting details

The semantics of the lifting and lowering operations is indicated in Fig. 13: The lifting and lowering operations in **Graph** simply return their argument. The more interesting case are the lifting and lowering operations in **ColoredGraph**, which override their corresponding implementations in **Graph**.

In `liftNode()`, a lookup in the map determines whether the same node has ever been lifted before. In this case, the corresponding instance of `ColoredGraph.Node` is directly returned. Otherwise, an instance of `ColoredGraph` that delegates to the node instance is created, stored in the map, and returned. The lookup in the map ensures that subsequent liftings for the same node yield the same `ColoredNode` wrapper.

The `lowerNode()` operation is the counterpart of `liftNode()`. It stores the `ColoredGraph` part of the node in the map and recursively asks its parent `super` to lower the parent of the node (`n.super`).

In the `liftNode()` operation, the parent object of the wrapper object is `super.liftNode(n)` rather than `n`, and in `lowerNode()`, the method returns `super.lowerNode(n.super)` rather than `n.super`. This ensures the mechanism will work when a class `C` overrides a virtual class that has already been overridden in the superclass of `C`. The anchor of the recursions are the `liftNode()` and `lowerNode` operations in the class that introduces the virtual class (in this case `Graph`, which simply return their argument, see Fig. 13).

What are the appropriate places to apply lifting and lowering operations? We think that the only reasonable solution is to apply it whenever hot state is evaluated; that is, the r-value of a hot instance variable in an expression `node` is actually `liftNode(node)`, and the l-value of a hot variable in an assignment `node = anExpression` is actually `node = lowerNode(anExpression)`. In our example, this means that the implementation of `getNode()` returns `this.liftNode(n)` rather than `n`, and the implementation of `setNode()` assigns the result of `lowerNode(n)` to `this.n`.

The calls to `liftNode()` and `lowerNode` are subject to late binding, because the respective implementations of `ColoredGraph` overrides the implementations in `Graph`. For example, the call `g.getNode()` in Fig. 12 yields a call to `Graph.liftNode()`, while the call `cg.getNode()` yields a call to `ColoredGraph.liftNode()`.

An important invariant of lifting and lowering is that the function combination `lowerV(liftV(v))` is the identity function, such that a statement like `node = node`, which translates to `node = lowerNode(liftNode(node))`, has the expected meaning.

This approach preserves static type safety because the lifting operation ensures that the evaluation of a hot instance variable yields an instance of the type which is appropriate for the respective context by dynamically creating and maintaining wrappers that delegate to the base objects. The hash table guarantees that we do not lose the state and identity of the individual parts of a delegation chain. Finally, the lowering operation guarantees consistency in the sense that all objects will only interact with other objects from the same family. For example, if we would execute the statement `g.setNode(new cg.Node())` with `g` and `cg` as in Fig. 12, and we would *not* apply lowering, we would suddenly have a colored node in a context `g` that does not assume color properties. A subsequent call like `g.getNode().setNeighbor(new g.Node())` would expose the

inconsistency because the original colored node would assume that its neighbor nodes would also be colored nodes.

However, this approach is much more than a fix to preserve type safety. Let us look at a more interesting example. Consider the code in Fig. 14. It shows a graph class which stores a graph as a list of nodes. A node has a list of incident edges and an edge stores its source and target node. The comments in the code indicate the places where the lifting and lowering actually takes place.

```
class Graph {
    Node[] nodes;
    void setNode(Node n, int i) {
        nodes[i] = n; // effectively assigns this.lowerNode(n)
    }
    Node getNode(int i) {
        return nodes[i]; // effectively returns this.liftNode(nodes[i])
    }

    virtual class Node {
        Edge[] edges;
        Edge getEdge(int i) {
            return edges[i]; // effectively returns this.liftEdge(edges[i])
        }
    }
    virtual class Edge {
        Node n1, n2;
        Node getTargetNode() {
            return n2; // effectively returns this.liftNode(n2)
        }
    }
}
class ColoredGraph extends Graph ... // as in Fig. 7
```

Fig. 14. A graph with hot state

Let us suppose we want to determine the chromatic number⁵ and/or a corresponding coloring for a specific graph. Let us further suppose we have an appropriate algorithm in a class `GraphColoring` as indicated in Fig. 15. Of course, the algorithm is directly applicable to any graph which has been instantiated as `ColoredGraph`. However, let us suppose that this is not the case for our sample graph because, say, we just want to know the chromatic number and are not interested in the coloring itself and do not want to waste the corresponding memory. Another reason might be that we want a graph that has different independent colorings with different meanings.

⁵ The minimum number of colors needed to color the vertices of a graph such that no two adjacent vertices have the same color.

```

class GraphColoring {
    int chromaticNumber(final ColoredGraph g) {
        ...
        g.Node node = g.getNode(i);
        node.color = Color.RED; // statically safe
        ...
    }
    void randomColoring(final ColoredGraph g) {
        ...
    }
}
// demo code
Graph g = ...;
GraphColoring coloring = new GraphColoring();
ColoredGraph cg1 = new ColoredGraph<g>();
int i = coloring.chromaticNumber(cg1);
...
ColoredGraph cg2 = new ColoredGraph<g>();
coloring.randomColoring(cg2);

```

Fig. 15. Independent on-the-fly extensions of a graph

The demo code in Fig. 15 shows how an arbitrary graph can be extended on-the-fly with the mechanisms of our approach. The color extension is only visible via `cg1` and `cg2`, respectively. The state and behavior of the graph remains unchanged if it is accessed via `g`. Please note how easy it is to create two completely independent colorings (chromatic and random coloring) for a specific graph instance. Due to subtype polymorphism, these extensions are also decoupled from the specific graph instance in the sense that `g` may also refer to an instance of `WeightedGraph` or even `ColoredGraph`. In the latter case, the extension would yield a coloring which is independent from the original coloring of `g`.

The last example in this section does not introduce new features but emphasizes two important properties of our approach: The ability to extend a collaboration which has already been extended (orthogonality), and transparent simultaneous behavior extensions for all objects of a collaboration instance.

Suppose we want to observe the progress of the coloring algorithm on the screen in case the respective graph is currently displayed. In other words, we want to be notified whenever the `setColor()` method is invoked for a node of that graph.

Consider the code in Fig. 16. It introduces an appropriate interface `ColorObserver` that is implemented by `GraphDisplay`. The class `NotifyingGraph` extends the behavior of all color nodes such that the `ColorObserver` is notified whenever the color of that node is changed. The demonstration code creates a `Graph g` and extends `g` to be a colored graph in the context of `cg1`. The colored graph `cg1` is again extended with the notifier behavior if the variable `screenDisplay` evaluates to `true`.

```

interface ColorObserver {
    void colorChanged(final ColoredGraph cg, cg.Node node, Color color);
}
class GraphDisplay implements ColorObserver { ... }
class NotifyingGraph extends ColoredGraph {
    ColorObserver o;
    public NotifyingGraph(ColorObserver o) { this.o = o; }

    override class Node {
        void setColor(Color color) {
            super.setColor(color);
            o.colorChanged(NotifyingGraph.this, this, color);
        }
    }
}
// demo code
Graph g = ...; GraphDisplay display = ...;
GraphColoring coloring = new GraphColoring();
ColoredGraph cg1 = new ColoredGraph<g>();
if (screenDisplay) cg1 = new NotifyingGraph<cg1>(display);
int i = coloring.chromaticNumber(cg1);

```

Fig. 16. Adding notifier functionality

Please note that the graph `cg1` that is potentially extended with the `NotifyingGraph` functionality is already an extended version of the original graph instance `g`.

The type of `cg1` is `ColoredGraph` and not `NotifyingGraph`. Nevertheless, the extensions defined by `NotifyingGraph` spread through all further actions via `cg1`. In particular, all `setColor()` invocations in the coloring algorithm are dispatched to the `setColor()` redefinition in `NotifyingGraph`, although the author of the coloring algorithm does not know anything about the existence of `NotifyingGraph`.

The powerful expressiveness of on-the-fly extensions is due to the fact that delegation layers allow simultaneous behavior extensions for *sets* of objects. To the best knowledge of the author, delegation layers are the first approach that enables such kind of operations.

7 Related Work

The relation to mixin layers [31], virtual classes [21], family polymorphism [12], and delegation [20, 18, 3, 27] has already been discussed in Sec. 1–4.

Java Layers [6, 9] are a Java-based implementation of mixin layers. Java Layers extend Java by supporting constrained parametric polymorphism and mixins. The authors acknowledge the composition consistency problem and propose different solutions (called *sibling pattern*), including a limited variant of virtual

types and a naming convention approach, to cope with this problem. An interesting approach in Java Layers, which might also be useful for delegation layers, is their notion of *deep conformance*, which extends Java's concept of interfaces to include nested interfaces.

Jiazzi [23] is a system that does also allow classes to be composed in a mixin layer style at compile time. *Jiazzi* is especially related to our work because it addresses both the composition consistency and the polymorphism problem (see Sec. 2). Their proposal for the composition consistency problem is based on the *open class pattern*, a kind of design pattern that mimicks the constructor semantics of virtual types. An application that uses a particular layer (*package* in the terminology of [23]) can be parameterized with different variants of this layer, thereby eliminating the polymorphism flaw of the original mixin layer idea. This is similar to the idea of parameterizing a method with a family object, as shown in Fig. 9. However, in contrast to delegation layers, composition and polymorphism in *Jiazzi* are pure compile-time / link-time concepts, there is no notion of subtyping polymorphism and subsumption among different variants of a layer.

In comparison with delegation layers, a practical advantage of all aforementioned compile-time approaches [31, 6, 9, 23] is that it is very much easier to create an efficient implementation with little or no runtime overhead.

In general, *virtual classes* are an interesting alternative or complement to parametric polymorphism. Please note that this is not the main focus of our approach, in contrast to the approaches in [32] and [7]. Therefore we do not introduce additional language means to express virtual classes defined outside the enclosing class, e.g., virtual classes like `StackItem` that are later overridden with `String` or `Point` in order to create a stack of strings or a stack of points.

Pluggable composite adapters (PCA) [26] are a language construct for on-the-fly adaptation of frameworks. A set of base objects can be dynamically extended with the functionality provided by a particular framework. The relations between base objects and framework objects are maintained by a lifting technique that is similar to the one proposed in this paper. However, in PCA, objects are lifted to types that are in general unrelated to their original type, whereas with delegation layers, objects are lifted to subtypes that delegate to the original object. In contrast to delegation layers, it is not possible to *change* the behavior of the lifted objects.

Delegation layers can also be seen as a form of *aspect-oriented programming* [17]. A delegation layer defines functionality that affects the behavior of a set of different classes and can thus be seen as a module for crosscutting concerns. In comparison with AOP languages like AspectJ [2], delegation layers have a very limited joinpoint model. On the other hand, delegation layers are much more dynamic than other AOP languages. For example, in AspectJ it would also be possible to extend the `Graph` class with color functionality. However, in this case *all* graphs would automatically be colored graphs; it would not be possible offhand to access a graph simultaneously both as a graph and a colored graph, or create independent colorings as in Fig. 15. The same argument applies to the

notification extension in Fig. 16. In AspectJ, the notification would automatically apply to *all* graphs and it would require additional measures (e.g., conditional statements in the form of `if (notifyEnabled) ...`) to be able to choose at runtime which graphs feature the notification behavior.

A number of approaches focus on the evolution of single objects or single classes. The basic idea of the *context relationship* [30] is that if a class `C` is context-related to a base class `B`, then `B`-objects can get their functionality dynamically altered by `C`-objects. A `C`-object may be explicitly attached to a `B`-object, or it may be implicitly attached to a group of `B`-objects for the duration of a method invocation. In *Rondo* [24], the behavior of single objects can be altered at runtime by means of so-called *adjustments*. With *predicate classes* [10], an object is automatically an instance of a predicate class whenever it satisfies a predicate expression associated with the predicate class. If an object is modified, the classification of an object can change, yielding in a different behavior of the object.

There have been a number of proposals related to *collaboration-* or *role-based design* [28, 4, 15, 29, 16, 25]. In contrast to these approaches, delegation layers focus on the definition and on-the-fly runtime combination of collaboration *variants*.

8 Summary and Future Work

In this paper, we proposed delegation layers, a new mechanism to define and combine sets of collaborating classes and objects. Since the modules to group such sets are classes and objects themselves, the concepts that proved so useful for single classes and objects - inheritance, delegation, late binding, instantiation, subtype polymorphism etc. - apply to sets of collaborating classes and objects as well.

Due to their strong runtime semantics, delegation layers are extremely flexible. In particular, the ability for local on-the-fly extensions, with which we can change the behavior of a *set* of objects (instead of a single object with classical delegation) seems to be very promising. We think that this is especially interesting with respect to the idea of aspect-oriented programming [17]. Current AOP approaches are frequently working on the basis of sophisticated source- or byte-code transformations and have little or no runtime semantics. We hope that delegation layers are a first step towards aspects with rich runtime semantics. We are currently working towards this goal.

Acknowledgements

We thank the anonymous reviewers for numerous helpful comments and Erik Ernst for insightful discussions which revealed serious shortcomings of previous drafts.

References

1. K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
2. AspectJ homepage, 2001. <http://aspectj.org>.
3. M. Büchi and W. Weck. Generic wrappers. In *Proceedings of ECOOP 2000, LNCS 1850*, pages 201–225. Springer, 2000.
4. K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *Proc. OOPSLA '89*, 1989.
5. G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices 25(10)*, pages 303–311, 1990.
6. A. Brown, R. Cardone, S. McDirmid, and C. Lin. Using mixins to build flexible widgets. In *1st International Conference on Aspect-Oriented Software Development AOSD '02*, 2002.
7. K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98*, 1998.
8. P. S. Canning, W. Cook, W. L. Hill, J. C. Mitchell, and W. G. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 273–280, 1989.
9. R. Cardone and C. Lin. Comparing frameworks and layered refinement. In *Proceedings of the 23rd International Conference on Software Engineering ICSE '01*, 2001.
10. C. Chambers. Predicate classes. In W. Olthoff, editor, *Proceedings ECCOP '93*, LNCS 707, pages 268–297. Springer, 1993.
11. M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1995.
12. E. Ernst. Family polymorphism. In *Proceedings of ECOOP '01*, LNCS 2072, pages 303–326. Springer, 2001.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
14. W. Harrison, H. Ossher, and P. Tarr. Using delegation for software and subject composition. Technical Report RC 20946(92722), IBM Research Division T.J. Watson Research Center, Aug 1997.
15. R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: Specifying behavioural compositions in object-oriented systems. In *Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices*, pages 169–180, 1990.
16. I. M. Holland. Specifying reusable components using contracts. In *Proceedings ECOOP '93, LNCS 615*, pages 287–308, 1992.
17. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings ECOOP'97*, LNCS 1241, pages 220–242, Jyväskylä, Finland, 1997. Springer-Verlag.
18. G. Kniesel. Type-safe delegation for run-time component adaptation. In R. Guerraoui, editor, *Proceedings of ECOOP '99*, LNCS 1628. Springer, 1999.
19. G. Kniesel. *Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, University of Bonn, Institute for Computer Science III, 2000.
20. H. Liebermann. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 1986.

21. O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89*. ACM SIGPLAN, 1989.
22. O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the Beta Programming Language*. Addison-Wesley Publishing Company, 1993.
23. S. McDirmid, M. Flatt, and W. Hsieh. Jiazz: New age components for old fashioned java. In *Proceedings of OOPSLA '01*, 2001.
24. M. Mezini. Dynamic object evolution without name collisions. In *Proceedings ECOOP '97, LNCS 1241*, pages 190–219. Springer, 1997.
25. M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, 1998.
26. M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters. In M. Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer, 2001. University of Twente, The Netherlands.
27. K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proceedings OOPSLA '01*, 2001.
28. T. Reenskaug. *Working with Objects: The OOram software Engineering Method*. Manning, 1995.
29. D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings OOPSLA '98*, 1998.
30. L. M. Seiter, J. Palsberg, and K. Lieberherr. Evolution of object behavior using context relations. *IEEE Transactions on Software Engineering*, 24:79–92, 1998.
31. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *Proceedings of ECOOP '98*, pages 550–570, 1998.
32. K. K. Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97*, 1997.
33. M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, 1998.
34. M. VanHilst and D. Notkin. Using role components to implement collaboration-based design. In *Proceedings OOPSLA 96*, 1996.