Implementing Reusable Collaborations with Delegation Layers

Klaus Ostermann
Siemens AG, Corporate Technology 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 proposed to express such slices in programming languages. We present a Java language extension that builds up on mixin layers and combines and generalizes virtual class and delegation concepts. The result is a language that allows runtime composition of components and better reusability of client code.

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. Application frameworks [9, 5] have been the first effort towards reusable collaborations. However, application frameworks have proven to be too inflexible for a number of reasons, see [14, 21, 15].On the design level, methodologies for *collaboration-* or *role-model based design* have been developed [2, 8, 17]. On the other hand, main stream programming languages have been equipped with light-weight linguistic means to group sets of related classes, e.g. name spaces in C++ or packages and inner classes in Java.

Mixin layers [18] are a more flexible implementation technique for collaborations. The key advantage over traditional framework techniques is (a) that they allow *sets* of classes to inherit from other *sets* of classes, that is, inheritance is scaled to a multi-class granularity, and (b) these sets of classes can be composed with each other by layering the sets and composing the corresponding classes in each layer with mixin-inheritance.

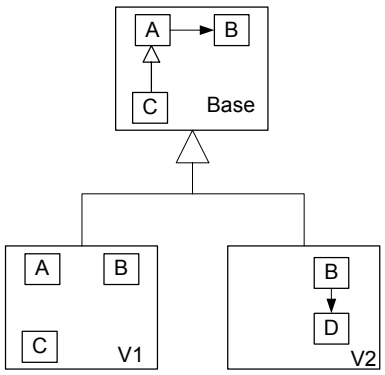

Figure 1: Collaboration inheritance

Fig. 1 shows sets of classes $V1$ and $V2$ that inherit from another set of classes `Base`. The `Base` collaboration defines classes `A`, `B` and `C`, and subcollaborations may extend or refine these classes (`V1`) or add additional classes (`V2`). For example, `Base` may encode a basic graph structure with classes `Node` and `Edge`, and `V1` and `V2` may be extensions of this basic collaborations that refine these classes to `ColoredNode` or `ColoredEdge`.

These subcollaborations can be combined freely. Fig. 2 demonstrates the semantics of a combination `V1(V2(Base))`: All inner classes are organized in a mixin style according to the definition of the outer abstractions. For example, we can easily create layers to mix-and-match the aforementioned different kinds of graphs. Fig. 3 shows the source code for C++ mixin layers `Graph`, `NodeColoredGraph`, and `EdgeColoredGraph`, which can be composed to a graph that has both colored nodes and colored edges (`NECG`) or only colored nodes (`NCG`).

Our model builts up on the concepts of mixin layers and extends it in two ways:

- **Runtime composition:** In contrast to the template technique by Smaragdakis and Batory we want to be able to compose our collaborations dynamically, that is, the desired sets of features should be determined at runtime.

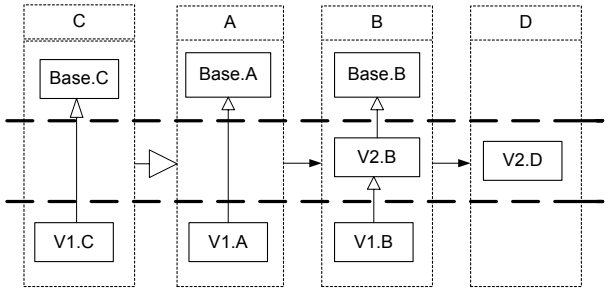

Figure 2: V1(V2(Base)):Layer combination with mixin-inheritance

```
class Graph {
  ...
  public:
  class Node { };
  class Edge { Node* n1, Node* n2; ... }
  Edge* connect(Node* n1,Node* n2) {
    return new Edge(n1,n2);
  }
  void addNode(Node *n) {...}
};
template <class Super>
class NodeColoredGraph : public Super {
public:
  class Node: public Super::Node {
    Color c;
    ...
  }
};
template <class Super>
class EdgeColoredGraph : public Super {
  class Edge: public Super::Edge {
    Color c;
    ...
  }
};
typedef
  NodeColoredGraph<EdgeColoredGraph<Graph> > NECG;
typedef NodeColoredGraph<Graph> NCG;
```

Figure 3: Graph Example with C++ mixin layers

```
class Client {
  void buildCompleteGraph(Graph *graph, int size) {
    ... Graph::Node *n = new Graph::Node; ...
    ... Graph::Edge *e = new Graph::Edge(n,m);
    ... graph->addNode(n); ...
  }
};
```

Figure 4: Client class that creates instances of the component classes

- **Client reusability**: We want to be able to use the same client code with different collaboration configurations. For this purpose, we introduce a notion of runtime polymorphism for these collaborations. This makes it possible to use existing client code (code that uses a composed collaboration, e.g. the aforementioned graph collaboration) with different collaborations at runtime. In general, this is not possible with mixin layers as presented in [18].

Delegation [11] can be seen as a generalization of mixin-based inheritance that allows to determine the inheritance hierarchy at runtime. For this reason, we think that *delegation layers* are a natural generalization of mixin layers, and they will be used to enable runtime composition.

The second point is more subtle and has to do with constructor calls. Constructors of a class `A` are called in code that creates instances of `A` and in constructors of subclasses. We distingiush two cases:

1. Constructor calls inside a collaboration, e.g., the constructor call inside `Graph::connect` in Fig. 3 creates an instance of `Edge`.
2. Constructor calls outside a collaboration, e.g., the method `Client::buildCompleteGraph` in Fig. 4 creates instances of `Node` and `Edge`.

In standard type systems, the binding of a constructor call to a specific constructor/class is static. This may be a problem: In the context of a compound collaboration as in Fig. 2, e.g., a `NECG` as in Fig. 3, we want the code in `Graph::connect` resp. `Client::buildCompleteGraph` to create an instance of the compound `Node` and `Edge` classes. However, with a usual type system, the constructor call in `Graph::connect` is statically bound to `Graph::Edge`, and the constructor call in `Client::buildCompleteGraph` is statically bound to the outer class that has to be annotated in the constructor call.

The same argument applies to calls to superclass constructors. Please note that in Fig. 2 the *compound* `C` inherits from the *compound* `A`. In the mixin layer approach, however, the compound `C` would be a subclass of `Base::A`.

This problem could be partly avoided by using *factory methods* [6] for the instantiation of the collaboration classes. In fact, factory methods can be seen as an insufficient simulation of virtual types because the usage of the factory methods cannot be enforced by the compiler, and important type information is lost whenever a factory method is used, leading to unsafe dynamic type casts. In addition, this does not work for constructor calls in subclasses as explained in the previous paragraph, because superclasses cannot be dynamically assigned in a language with static inheritance.

We think that this problem can be seen as a variant of the *self problem* [11], a.k.a. *broken delegation* [7]: In a composite component, all actions should be applied to the composite component, rather than to an individual part of it. In

```
class A {
  void foo() { print("A"); }
}
class B extends A {
  void foo() { print("B"); super.foo();}
}
class C extends A {
  void foo() { print("C"); super.foo();}
}
...
A a = new C();
a.foo(); // prints "CA"
A a = new C<new B()>();
a.foo(); // prints "CBA"
```

Figure 5: Code example for delegation

the original formulation of the self problem, this refers to method calls; in our case, this refers to constructor calls.

Smaragdakis and Batory acknowledged this problem and proposed to use *virtual types* to cope with it. Virtual types are a concept from the Beta programming language [13]. A virtual type can be refined by superclasses in an inheritance chain and the most refined version is the one used by the superclass code. However, further details about their notion of virtual types are not provided.

We try to fill this gap and combine virtual types with a notion of type variables as recently presented by Erik Ernst [4]. The important difference of his *family polymorphism* approach to the previous virtual types approaches is that a virtual type is a type variable of objects of the enclosing class. These type variables are valid type annotations in the program; objects of the enclosing class can be seen as a repository of types. Family polymorphism is especially useful in our context because it guarantees statically safe, consistent usage of a collaboration and does not need runtime type checks as in [19] and [12]. By using type variables, many programs can be proved statically safe without employing final bindings [20] or type exact variables [3].

2. DELEGATION LAYERS AND VIRTUAL CLASSES

In the first step we add a restricted variant of delegation to Java, and in the second step we generalize this notion to multi-class granularity by adding virtual classes.

To make the discussion simple, we restrict our notion of delegation to *static delegation*. This means that parent objects can be initialized at runtime, but once these references are initialized, they cannot be changed, similar to a `final` variable in Java. This restricted variant has also been employed by Büchi and Weck [1], who in addition add the notion of *transparency*, meaning that an object is a subtype of the *dynamic* type of its parent (in contrast to other proposals like [10] and [16][1]). In the context of delegation layers we use delegation with transparency.

[1]In models that support full dynamic delegation there are good reasons for not supporting transparency

```
class Base {
  virtual class A {
    B b;
    void foo() { b = new B(); }
  }
  virtual class B { ... }
  virtual class C extends A { ...}
}

class V1 extends Base {
  override class A { ... }
  override class B { ... }
  override class C { ... }
}

class V2 extends Base{
  override class B {
    D d;
    ...
  }
  override class D { ... }
}

class Client {
  void bar(Base base) {
    B b = new base.B(); // creates V1::B
  }
}

...
main() {
  Base base = new V1<new V2()>();
  new Client().bar(base);
  V2 = (V2) base; // succeeding dynamic cast
}
```

Figure 6: Delegation layers

We unify standard inheritance and our restricted variant of delegation as follows: In a `new C()` expression for a class `C extends A` we may *optionally* specify a parent object (delimited by `<>`) that has to be a subtype of `A`. For example, let `B` be a subclass of `A`. Then `new C()` creates an instance of `C` with superclass/parent `A` (usual semantics), and `new C<new B()>()` creates an instance of `C` with superclass/parent `B` (see Fig. 5). For further details about the semantics of delegation we refer to the existing approaches, e.g., [10, 1, 16].

So far, nothing really exciting happened. The crucial point is that we extend the delegation semantics recursively to nested classes by incorporating virtual classes in the sense of [4], that is, with virtual classes as type variables of objects.

We annotate an inner class as a virtual class with the `virtual` modifier (corresponds to `:<` in Beta). A class that overrides a virtual class is annotated with the `override` keyword (corresponds to `::<` in Beta). If an inner class `A` of a class `Base` is annotated as `virtual`, this means, that `Base.A`

```
class Base {
  virtual class A {
    Base.this.B b;
    void foo() { b = new Base.this.B(); }
  }
  virtual class B { ... }
  virtual class C extends this.A { ...}
}

class V1 extends Base {
  override class A extends super.A { ... }
  override class B extends super.B { ... }
  override class C extends super.C { ... }
}

class V2 extends Base{
  override class B extends super.B {
    V2.this.D d;
    ...
  }
  virtual class D { ... }
}
```

Figure 7: Semantics of delegation layers

is no longer a valid type specifier because `A` is a type variable of *objects* of type `Base`. Instead, `base.A` is a valid type specifier, whereas `base` is a reference of type `Base`.

Delegation and virtual classes work together as follows: **The superclass of a class `V1.A` that overrides another class `Base.A` is the *virtual superclass* `super.A`.** Please note that in the case of delegation `super` is a reference to the parent object. This combination of delegation and virtual classes allows us to combine our collaboration classes at runtime with the composition semantics illustrated in Fig. 2.

Fig. 6 shows source code that corresponds to Fig. 1 and uses these features. The classes `Base`, `V1` and `V2` specify slices that can be composed. The `main()` method composes `V1` with `V2` dynamically. The semantics of the composition `new V1<new V2()>()` becomes clear in Fig. 7. It shows the same code as in Fig. 6 but it makes the implicit type scoping explicit: All type annotations that correspond to virtual classes are type annotations relative to an object reference. The internal behavior of the resulting compound collaboration is similar to the one schematically presented in Fig. 2: The different parts are layered on each other and combined in a mixin-style.

Fig. 8 rewrites the functionality in Fig. 3 using our model. The advantages of this rewritten version become apparent if we compare the "main functions" for these pieces of code (Fig. 9 and 10):

- We are able to compose our layers at runtime by using delegation. Due to subtype polymorphism, we do not even need to know exactly which components participate in the composition. For example, instead of creating the instance of `NodeColoredGraph` in Fig. 9

```
class Graph {
  List nodes;
  virtual class Node { ... }
  virtual class Edge {
    Node n1,n2;  ...
  }
  Edge connect(Node n, Node m) {
    return new Edge(n,m);
}
class NodeColoredGraph extends Graph {
  override class Node {
    Color c;
    ...
  }
}
class EdgeColoredGraph extends Graph {
  override class Edge {
    Color c;
    ...
  }
}
class Client {
  void buildCompleteGraph(Graph graph, int size) {
    ... graph.Node n = new graph.Node() ...
    ... graph.Edge e = new graph.Edge(n,m); ...
    ... graph.addNode(n); ...
  }
}
```

Figure 8: Graph variations with delegation layers and virtual classes

```
...
main() {
  EdgeColoredGraph g =
     new EdgeColoredGraph<new NodeColoredGraph()>();
  Graph g2 = g; // statically safe
  new Client().buildCompleteGraph(g2,5);
  NodeColoredGraph g3 =
     (NodeColoredGraph) g; // succeeding dynamic cast
}
```

Figure 9: Main method for the source code in Fig. 8

```
...
main() {
  NECG *g = new NECG;
  Graph *g2 = g; // statically safe
  // the next call will not work properly
  (new Client)->buildCompleteGraph(g2,5);
  // NCG *g3 = g; rejected by the compiler
}
```

Figure 10: Main method for the source code in Fig. 3 and 4

we could have composed `EdgeColoredGraph` with some unknown layer that has been passed as a parameter of type `Graph`.

- The client code can be used with all kinds of graphs. The virtual type mechanism assures that the client creates the "right" instances of the nested classes, regardless of the particular collaboration combination. For example, in the control flow of `main()`, instances of `ColoredNode` and `ColoredEdge` are created in `buildCompleteGraph()`. In contrast, the constructor calls in the `buildCompleteGraph` method in Fig. 4 are statically bound to the inner classes of `Graph`.

- The aforementioned notion of transparency enables us to access the features of all layers that constitute a component. For example, the graph reference in Fig. 9 can be dynamically casted to `NodeColoredGraph`. In contrast, an equivalent type conversion in Fig. 10 is rejected by the compiler.

3. CONCLUSIONS AND FUTURE WORK

We presented a Java language extension that builds up on mixin layers and combines virtual class and delegation concepts. The resulting model is more flexible than previous approaches and allows better reuse of client code.

However, at the time of writing, this is a work in progress, and there are several open issues. The combination of delegation and virtual types with type variables has some typing implications not explictly addressed so far. The point is that the same layer instance may be used in different contexts, e.g., there may be direct references to an instance of `NodeColoredGraph`, and it may be simultaneously used as a parent of a `EdgeColoredGraph`. This would mean that the type variable would have a different value, depending on the context in which this layer instance is used. Without further restrictions, this would be unsound, because instance variables, that are shared among all contexts, would have a different type, depending on the context in which they are accessed.

One solution to cope with this problem would be to forbid such unsound sharing. However, we think that this problem is due to the fact that virtual types intermix two different concepts. The first concept is the typing part of virtual types: With the typing part of virtual types, more programs can be verified to be statically type safe, but this part of virtual types is *passive*: The semantics of the program is not changed. The second concept is the *active* part of virtual types, namely the redirection of constructor calls. This part of the virtual types concept changes the control and data flow of the program and is independent of the typing part.

We feel that each of these concepts is useful on its own and thus seek for a model that separates these notions. We hope that this allows us to cope with the aforementioned problem because the active part of virtual types has no typing implications at all.

An interesting issue that is also related to typing is related to Java's separation of classes and interfaces. It would be desirable to be able to specify interfaces for our collaborations, so that client code can access a particular collaboration via a well-defined interface. We would need some special tag that could be used to mark an inner interfaces in Java as a virtual, e.g., the keyword `virtual`. The semantics is that all classes that implement this interface have to create a virtual class with the same name. For example, an interface for the graph collaboration would look like this:

```
interface GraphInterface {
  virtual interface Node { ... }
  virtual interface Edge { ... }
}
class Graph implements GraphInterface {...}

class Client {
  void buildCompleteGraph(GraphInterface graph,
                          int size ) {
    ... new graph.Node() ...
    ...
  }
}
```

Some additional notational means are needed to cope with constructors that require arguments. For example, we may allow to introduce a "constructor interface" into such virtual interfaces. Conformance of the constructors of implementing virtual classes can be checked easily at compile time.

4. REFERENCES

[1] M. Büchi and W. Weck. Generic wrappers. In *Proceedings of ECOOP 2000, LNCS 1850*, pages 201–225. Springer, 2000.

[2] K. Beck and W. Cunningham. A laboratory for object-oriented thinking, 1989.

[3] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98*, 1998.

[4] E. Ernst. Family polymorphism. In *Proceedings of ECOOP '01*, 2001.

[5] M. Fayad, D. Schmidt, and R. Johnson. *Building Application Frameworks*. Wiley, 1999.

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

[7] 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.

[8] 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.

[9] R. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[10] G. Kniesel. Type-safe delegation for run-time component adaptation. In R. Guerraoui, editor, *Proceedings of ECOOP '99*, LNCS 1628. Springer, 1999.

[11] H. Liebermann. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 1986.

[12] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings of OOPSLA '89*. ACM SIGPLAN, 1989.

[13] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the Beta Programming Language*. Addison-Wesley Publishing Company, 1993.

[14] M. Mattson, J. Bosch, and M. E. Fayad. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10), October 1999.

[15] 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, 2000. University of Twente, The Netherlands.

[16] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proceedings OOPSLA '01*, 2001.

[17] D. Riehle and T. Gross. Role model based framework design and integration. In *Proceedings OOPSLA '98*, 1998.

[18] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *Proceedings of ECOOP '98*, pages 550–570, 1998.

[19] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97*, 1997.

[20] M. Torgersen. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages*, 1998.

[21] M. VanHilst and D. Notkin. Using role components to implement collaboration-based design. In *Proceedings OOPSLA 96*, 1996.