



HAL
open science

Empowering Collections with Swarm Behavior

Adrian Kuhn, David Erni, Marcus Denker

► **To cite this version:**

Adrian Kuhn, David Erni, Marcus Denker. Empowering Collections with Swarm Behavior. 2010. inria-00523507

HAL Id: inria-00523507

<https://inria.hal.science/inria-00523507v1>

Preprint submitted on 25 Sep 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Empowering Collections with Swarm Behavior

Adrian Kuhn David Erni Marcus Denker

Software Composition Group
University of Bern
www.iam.unibe.ch/~scg

Abstract. Often, when modelling a system there are properties and operations that are related to a group of objects rather than to a single object. In this paper we extend Java with *Swarm Behavior*, a new composition operator that associates behavior with a collection of instances. The lookup resolution of swarm behavior is based on the element type of a collection and is thus orthogonal to the collection hierarchy.

1 Introduction

Doug Lea wrote in 1994 [8]: “Evidence from over a decade of experience in (non-OO) distributed systems, especially, suggests that groups will become central organizing constructs in the development of large OO systems”. Almost 15 years later, many object-oriented languages still lack dedicated first-class concepts to model the behavior of groups. In existing methodologies and languages, methods are either associated with a class as a whole or with a single instance. However, groups of instances often possess discernible behavior that is not well-captured by current concepts and notations.

Throughout this paper we use a running example to motivate and explain group behavior. Figure 1 illustrates SWARM, a small game where creatures fight each other. Each creature is modeled as an object. Creatures have hitpoints and can attack each other by sending messages.



Fig. 1. In the SWARM game, a collection of fish is under attack by a shark. On the left: the collection of fish cannot defeat the attacking shark. On the right: the collection of fish uses *Swarm Behavior* to defeat the attacking shark.

The user interface of SWARM is rather simple. Players select message from pop-up menus to attack opponents. Clicking on a single selected creature opens a pop-up menu with all messages understood by that creature—*but how shall the game deal with multiple selections?*

Modern user interfaces address multiple selection using aggregation and projection. Given a group of selected creatures, the pop-up menu is populated with the intersection of messages that are understood by all creatures in the group (aggregation). When a menu item is selected the according message is sent to all creatures in the group (projection).

However the behavior achieved through the use of aggregation and projection is limited by the behavior of single elements. Complex behavior cannot be modelled using aggregation and projection alone. [Figure 1](#) uses a metaphor to illustrate the difference between simple aggregation and complex group interaction: the collection of fish on the left cannot defeat the attacking shark whereas the collection of fish on the right uses "swarm behavior" to successfully defeat the attacking shark.

The SWARM game might seem an unrealistic example. However, the addressed problem is real in modeling software. There are many realistic applications that would benefit from swarm behavior. The authors of this paper have first come across the missing group-related abstractions when designing the Moose reengineering tool suite, where 20% of the domain code could be refactored towards Swarm Behavior [\[7\]](#). Other applications that would benefit from swarm behavior are file navigators such as the Finder of OSX or any other application that allows multiple selection of domain items. In general any application whose domain model includes grouping could benefit from Swarm Behavior. Just think how often you have come across classes such as `CustomerList` or `AccountList` that extend a general-purpose collection class with behavior specific to their corresponding domain objects.

In this paper we propose *Swarm Behavior*, a new composition operator that associates behavior with a collection of instances. The lookup of swarm behavior is based on the element type of a collection and is thus orthogonal to the collection hierarchy.

The contributions of this paper are:

- We report and analyze the shortcomings of current group-related idioms.
- We propose *Swarm Behavior*, a new composition operator that associates the behavior of collection instances with the type of their elements.
- We present JAVAGROUPS [\[5\]](#), a research prototype that extends Java's OpenJDK compiler with Swarm Behavior.

The remainder of this paper is structured as follows: [Section 2](#) reports on the shortcomings of current group-related idioms. [Section 3](#) presents the model of Swarm Behavior. [Section 4](#) presents our prototype implementation for Java. [Section 5](#) provides the details of how we extended the Java compiler. [Section 6](#) discusses issues related to Swarm Behavior in general and to the JAVAGROUPS prototype. [Section 7](#) discusses related work and [Section 8](#) concludes the paper.

2 Group-related Idioms in Current Languages

In this section we report and analyze the shortcomings of current group-related idioms. We present three common idioms that are often used in programming languages (such as *e.g.* Java, Smalltalk, and Ruby) that lack first-class support for group behavior. For each idiom, we implement the `swarmAttack` method of the SWARM example. We examine each implementation and eventually conclude the properties of a possible solution for group-related behavior. We use the following criteria to examine the implementations:

- In which lexical scope is the `swarmAttack` method implemented?
- For which subtype of `Collection` is the `swarmAttack` available?
- Is the availability of `swarmAttack` restricted to collections of `Fish` instances?
- Given a subclass of `Fish`, can it override the behavior of `swarmAttack`?

2.1 Idiom #1, Element-specific Subclass of Collection

This idiom subclasses a general-purpose collection class, limits the type of the collection elements and implements behavior specific for groups of this type. The following source code creates a subclass of `ArrayList` and implements the `swarmAttack` method.

```
// define element specific subclass
public class FishSwarm extends ArrayList<Fish> {
    public void swarmAttack(Creature creature) {
        int swarmSize = this.size();
        creature.damage(swarmSize * swarmSize);
    }
}
// Create swarm and attack
Creature shark = new Shark();
FishSwarm swarm = new FishSwarm();
for (int n = 0; n < 10; n++) swarm.add(new Fish());
swarm.swarmAttack(shark);
```

Examination: This solution is fatally tied to a concrete collection subtype. The swarm behavior neither is available for other collection types, such as linked lists or tree sets, nor is it available on array lists created by third-party code.

2.2 Idiom #2, Re-open the Collection Hierarchy

This idiom is only applicable for languages that can re-open classes. The idiom extends the root of the collection hierarchy with a `swarmAttack` method. Since Java cannot re-open classes, we provide a Ruby example.

```
// re-open array class and append new method
class Array
    def swarm_attack(creature)
        swarm_size = self.size()
```

```

    creature.damage( swarm_size * swarm_size )
  end
end
// Create swarm and attack
shark = Shark.new()
swarm = [ Fish.new(), Fish.new(), Fish.new() ]
swarm.swarm_attack(shark)

```

Examination: Since the swarm behavior is implemented at the root of the collection hierarchy we can possibly invoke the swarm behavior on collections of the wrong element type.

2.3 Idiom #3, Static Helper Method

This idiom uses a static helper method that expects the collection as its parameter. Static helper methods are typically defined in the scope of the element class. The listing below defines a static helper method `swarmAttack` that expects an object of type `Collection<Fish>` as first parameter.

```

// defines a static helper method
public class Fish extends Creature {
    public static void swarmAttack(Collection<? extends Fish> swarm,
        Creature creature) {
        int swarmSize = swarm.size();
        creature.damage(swarmSize * swarmSize);
    }
}
// Create swarm and attack
Creature shark = new Shark();
Collection<Fish> swarm = new ArrayList<Fish>();
for (int n = 0; n < 10; n++) swarm.add(new Fish());
Fish.swarmAttack(swarm, shark);

```

Examination: This solution is flawed by the lack of polymorphism. Given a class `Herring` that subclasses `fish`, group methods of `herring` can neither inherit nor overload group methods of `fish`. There is not inheritance hierarchy of swarm behavior that would parallel the inheritance hierarchy of the element types.

2.4 Conclusion of our Analysis

We conclude our analysis of the three idioms in a table. Beside the three idioms discussed above, the table also includes the *Swarm Behavior* that we present in the next section as novel composition operator for group-related behavior.

	#1	#2	#3	Swarm Behavior
Defined in lexical scope of <code>Fish</code> .	–	–	yes	yes
Available for all subtypes of <code>Collection</code> .	–	yes	yes	yes
Limited to collections of <code>Fish</code> instances.	yes	–	yes	yes
Subclasses can override <code>swarmAttack</code> .	yes	(yes)	–	yes

3 The Model of Swarm Behavior

In this chapter we propose Swarm Behavior as a new composition operator that aims to avoid the limitations of the idioms in [Section 2](#) while combining their strengths. The listing below implements `swarmAttack` as Swarm Behavior of `Fish`. An annotation is used to indicate that `swarmAttack` is a group method rather than an instance method.

```
public class Fish extends Creature {
    public void attack(Creature creature) {
        creature.damage( 0 );
    }
    @Group
    public void swarmAttack(Creature creature) {
        int swarmSize = this.size();
        creature.damage( swarmSize * swarmSize );
    }
}
```

Please note, that within the body of `swarmAttack` the pseudovariable `this` is bound to a collection of fish instances rather than a single fish instance. It is thus save to invokes the collection's `size` method.

The following listing illustrates a call site of the above group method. Group methods are invoked on a collection of their class's instances.

```
// create a swarm and attack
Collection<Fish> swarm = new ArrayList<Fish>();
for (int n = 0; n < 10; n++) swarm.add(new Fish());
Shark shark = new Shark();
swarm.swarmAttack(shark);
```

Please recall that `swarmAttack` is defined within the lexical scope of `Fish` but invoked on a collection of fish instances rather than a single instance. We say that `swarm` is a *container* that contains *elements* of type `Fish` and method `swarmAttack` is a *group method* of `Fish`.

[Figure 2](#) illustrates the object model of class `T` involving swarm behavior. Swarm Behavior extends the common object model and adds a new option to organize methods. This is achieved by extending the object model with a new kind of methods. In addition to instance and static methods, we allow classes to define group methods. Swarm behavior changes the way method lookup works for any instance of `Collection` or subclass thereof. The method lookup is extended to take into account both the collection's class hierarchy and the element type. In the same way lookup of `this` within the lexical scope of group methods is changed.

The procedure `SWARM-LOOKUP(O, S)` defined the lookup of *Swarm Behavior* as follows. The procedure has two input parameters, a receiver O and a method selector S , and either returns a method m or fails.

SWARM-LOOKUP(O, S)

```

1  ▷ Lookup of instance methods.
2   $C_I \leftarrow \text{class}[O]$ 
3  while  $C_I \neq \perp$ 
4      do  $\mathcal{M}_I \leftarrow \text{instance-methods}[C_I]$ 
5          if  $M_I \in \mathcal{M}_I$  and  $\text{selector}[M_I] = S$ 
6              do return  $M_I$ 
7           $C_I \leftarrow \text{superclass}[C_I]$ 
8  ▷ Fail unless receiver is a collection.
9  if  $\neg(O \text{ instance-of } \text{Collection})$  return FAIL
10 ▷ Lookup of group methods.
11  $\mathcal{C}_E \leftarrow \{C : C = \text{class}[E] \wedge E \in \text{elements}[O]\}$ 
12  $C_S \leftarrow \text{LEAST-UPPER-BOUND}(\mathcal{C}_E)$ 
13 while  $C_S \neq \perp$ 
14     do  $\mathcal{M}_S \leftarrow \text{group-methods}[C_S]$ 
15         if  $M_S \in \mathcal{M}_S$  and  $\text{selector}[M_S] = S$ 
16             do return  $M_S$ 
17          $C_S \leftarrow \text{superclass}[C_S]$ 
18 return FAIL

```

SWARM-LOOKUP works as follows: Lines 2–7 perform an instance lookup in the same way as would be done without swarm behavior. The instance lookup walks up the superclass chain of the receiver’s class and returns if a method that matches the method selector S is found. Line 9 stops the lookup unless the receiver O is an instance of `Collection`. Lines 11–12 get the element type of the receiving collection O . The element type of a collection is defined as the most specific superclass of all elements in the collection. Lines 13–17 eventually perform the lookup of group methods. Group lookup works the same as instance lookup but differs twofold 1) group lookup starts at the element type C_S rather than at the receivers class, and 2) group lookup inspects the *group* methods of class rather than the instance methods.

Since by this definition, the group methods of a collection are hidden by its instance methods, an escape is required to invoke shadowed group methods. Imagine that the implementation of `swarmAttack` would invoked `Fish.this.size()` instead of `this.size()`, in this case, the lookup of `size` would skip the instance lookup and directly start the group lookup with `Fish` as group class.

The LEAST-UPPER-BOUND procedure is defined as follows. Given a collection \mathcal{C} of classes, the procedure returns the most specific superclass of the input classes. (It is assumed that all classes have one root superclass in common).

LEAST-UPPER-BOUND(\mathcal{C})

```

1   $C_0 \leftarrow$  any element of  $\mathcal{C}$ 
2  for each  $C \in \mathcal{C}$ 
3      do while  $C_0 \triangleleft C$ 
4          do  $C_0 \leftarrow \text{superclass}[C_0]$ 
5  return  $C_0$ 

```

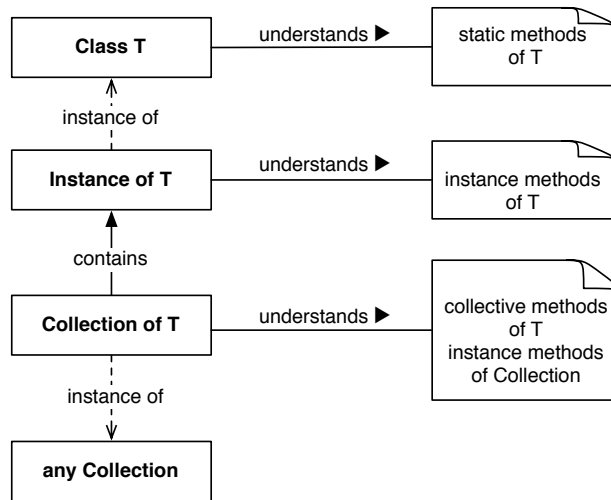


Fig. 2. The object model of a class T with swarm behavior. Group methods are defined as part of T’s model and invocable on any collection of T instances. Neither are group methods invocable on single instance of type T or nor are they the same as class methods of T.

4 Extending Java with Swarm Behavior

In this section we present JAVAGROUPS [5] a proof-of-concept prototype of swarm behavior. Our implementation extends the Java compiler¹ with support for swarm behavior. In this section we discuss how JAVAGROUPS modifies the Java compiler and how the group lookup is achieved. JAVAGROUPS takes a lightweight approach. We hook into the Java compiler and apply source code transformation such that swarm-enabled source code, which may contain semantically invalid constructs, is transformed to valid java code.

4.1 Example of JAVAGROUPS Transformations

Let’s consider the example given earlier in Section 3, that invokes a group method on a collection of fish. This is invalid Java code as the **Collection** does not implement the called method **swarmAttack**, hence JAVAGROUPS must transform this code as follows:

```

// before transformation
Collection<Fish> swarm = new ArrayList<Fish>();
Shark shark = new Shark();
swarm.swarmAttack(shark);

```

¹ The Java compiler is open source and available as part of the OpenJDK project: <http://www.openjdk.org/groups/compiler>.


```

// after transformation
Collection<Fish> swarm = new ArrayList<Fish>();
Shark shark = new Shark();
new Fish.Fish$Group(swarm).swarmAttack(shark);

```

The transformed code differs only in the last line. Instead of calling `swarmAttack` directly on `swarm`, the `swarm` collection is wrapped in an element specific subclass of `Collection` that implements the group method. The transformed code uses a variation of the “Element-specific Subclass of Collection” idiom given in [Section 2](#). We avoid the subclassing limitation of this idiom by creating a proxy that wraps a collection instance. This wrapper class has been generated by `JAVAGROUPS` in a previous compiler phase and is described in the following.

In the following listing the class `Fish` is given, it implements `swarmAttack` as a group method. `JAVAGROUPS` transforms this code as follows:

```

// before transformation
public class Fish extends Creature {
    @Group public void swarmAttack(Creature creature) {
        int swarmSize = this.size();
        creature.damage( swarmSize * swarmSize );
    }
}

// after transformation
public class Fish extends Creature {
    public static class Fish$Group <E extends Fish> implements
        Collection<E> {

        protected Collection<E> delegate;
        public Fish$Group(Collection<E> delegate){
            this.delegate = delegate;
        }
        public void swarmAttack(Creature creature) {
            int swarmSize = this.size();
            creature.damage( swarmSize * swarmSize );
        }
        // delegation
        public boolean add(E param0){
            return this.delegate.add(param0);
        }
        // remaining delegation methods
        ...
    }
}

```

The transformation moves all methods of a class that are tagged with `@Group` into a separate class called `Fish$Group`. (Out of convenience, *i.e.* namespacing and ease of compiler extension, we decided to use a static inner class. Static inner classes are, despite their name, not inner classes but rather nested top level classes.) `Fish$Group` is a proxy that wraps a collection instance. `Fish$Group`

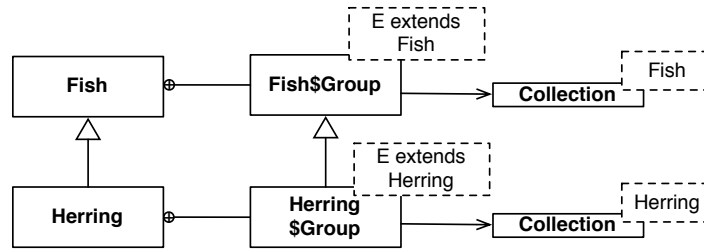


Fig. 3. Given two example classes `Fish` and `Herring` extending `Fish`. For both classes a wrapper class is generated and parametrized with its corresponding element type. `Herring`'s wrapper class extends `Fish`'s wrapper class just like `Herring` extends `Fish`. Both wrapper classes delegate to a parametrized `Collection`.

implements all group methods of `Fish` as well as the complete interface of `Collection`. The implementation of the interface simply delegates any calls to the enclosed collection instance.

Please note that, even though swarm behavior rebinds `this` within group methods, the transformer does not alter any `this` statement. The `this` statements are rebound from the fish class to the wrapper class as a side effect of being moved from fish's lexical scope to the scope of the wrapper class.

At this point, the transformation is complete since the example does not make use of inheritance. Inherited group methods require additional transformation which are not discussed here due to space restrictions. For more details, please refer to the `JAVAGROUPS` documentation [5].

4.2 Inheritance among Group Methods

Inheritance among wrapper classes is modeled by paralleling the inheritance hierarchy of element classes among the synthetic wrapper classes. Figure 3 shows two example classes `Fish` and `Herring`, that extends `Fish`. Both classes have a corresponding wrapper class that is parametrized with the corresponding element type. The type parameter of the wrapper class is set to `E extends T`. `Herring`'s wrapper class extends `Fish`'s wrapper class because `Herring` extends `Fish`, and both of them define group methods. Both wrapper classes delegate to `Collections` that contain instances of their corresponding elements.

Figure 4 illustrates a sequence diagram of group method lookup on an instance of `Container`. Assume we call a method on a `Collection` of `Herrings`. The selected method is a group method defined for a group of `Fish`, and hence it is available for a group of `Herrings`, too. First, we check if the method is a method available for instances of `Collection`. This is not the case, so we start the lookup for group methods in `Herring` as the container (in this case `Collection`) is instantiated with the type parameter `Herring`. If the group method is not found for `Herring`, continue with `Herring`'s supertype, in this case `Fish`. In that example, the group method is found in `Fish`'s type and hence

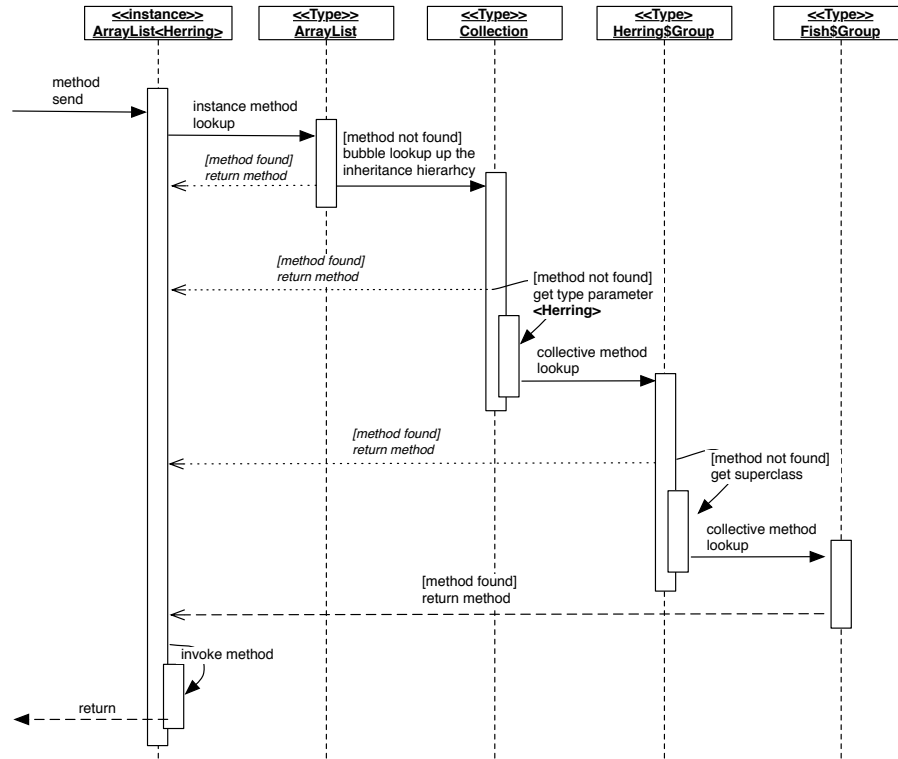


Fig. 4. Example method invocation on a Collection of Herrings. Let the selected method be a group method defined on Fish. After the common lookup failed we start the lookup for group methods in Herring. Finally the group method is found on Fish, returned, and invoked.³

lookup is successful. The dotted lines indicate where the lookup process can stop if a method is found. The dashed lines indicate where the lookup stops in this concrete example.

5 Modification to the Java Compiler

This section provides the details of how we extended the Java compiler with `JAVAGROUPS`. Readers not familiar with the Java compiler may decide to skip this section.

We start with a brief overview the Java compiler first and give an enumeration of its main compiler phases. [Figure 5](#) illustrates the passes that are performed and

³ [Figure 4](#) makes use of combined fragments (an UML 2.0 extension), conditional flow of sequence is indicated using dotted lines with conditions given in brackets.

highlights the points where JAVAGROUPS differs. The OpenJDK Java compiler passes these phases

1. **parse:** Reads a set of *.java source files and maps the resulting token sequence into Abstract Syntax Tree (AST) Nodes.
2. **enter:** Enters symbols for the definitions into the symbol table.
3. **process annotations:** If requested, processes annotations found in the specified compilation units.
4. **attribute:** Attributes the syntax trees. This step includes name resolution, type checking and constant folding.
5. **flow:** Performs dataflow analysis on the trees from the previous step. This includes checks for assignments and reachability.
6. **desugar:** Rewrites the AST and translates away some syntactic sugar.
7. **generate:** Generates source files or class files.

5.1 Transformations to Generate Synthetic Wrapper Classes

Most of the AST transformation is done during the compiler's Enter pass. In step 1a), JAVAGROUPS transforms the syntax tree by adding the wrapper class as inner class of its corresponding element. In step 1b) JAVAGROUPS scans the generated classes again to add inheritance information to them.

5.2 Generate Wrapper Classes

Let \mathcal{C} be any public toplevel class that gets compiled. We define:

$$\mathcal{M}(\mathcal{C}) = \{m \mid m \text{ is a method of } \mathcal{C}\}$$

$$\mathcal{G}(\mathcal{C}) = \{m \mid m \in \mathcal{M}(\mathcal{C}) \wedge m \text{ is annotated with } @Group \}$$

Before the compiler starts the Enter phase, JAVAGROUPS performs the following transformations for each class \mathcal{C} .

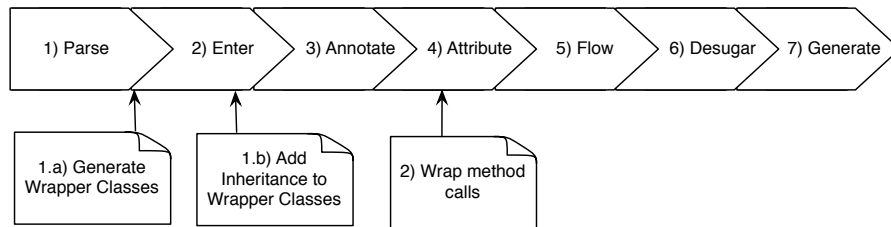


Fig. 5. Before the Compiler starts the Enter pass, JAVAGROUPS generates the wrapper classes. Later on during enter, JAVAGROUPS adds inheritance information to the previously generated wrapper classes. During attribution, JAVAGROUPS wraps the method calls to group methods into the wrapper classes.

- **add an inner class:** For each class \mathcal{C} where $\mathcal{G}(\mathcal{C}) \neq \emptyset$ a wrapper class $\mathcal{C}_{\text{group}}$ is generated. This class is added to \mathcal{C} 's definitions as inner class. Its name is composed of the string "Group" followed by a dollar sign "\$" followed by the class name of the associated toplevel class. The resulting classname for that inner class $\mathcal{C}_{\text{group}}$ is $\mathcal{C}\$Group$.
- **creating a constructor:** Generate a constructor \mathcal{K} for $\mathcal{C}_{\text{group}}$. As described in [Section 4](#) the constructor takes a `Collection<C>` as parameter. This parameter is later used to refer to the original collection for delegation.
- **moving group methods:** Now, each method in $\mathcal{G}(\mathcal{C})$ is moved to $\mathcal{C}_{\text{group}}$. This happens by removing it from the definitions of \mathcal{C} , and appending it to the definition of $\mathcal{C}_{\text{group}}$. Furthermore, the AST of those methods is transformed to use generics. This means that every occurrence of \mathcal{C} as identifier is replaced by the name of the type parameter used for generics, *i.e.* `E`.
- **generating methods from the collection interface:** The final step for this pass is to generate the methods needed for delegation. Let \mathcal{F} be the containers interface. In our implementation \mathcal{F} is equal to the interface `Collection`. Furthermore, let $\mathcal{I}(\mathcal{F}) = \{m \mid m \text{ is defined by } \mathcal{F}\}$. Now the set

$$\mathcal{D}(m) = \{m \text{ delegates to the Collection passed to } \mathcal{K}\}$$

$$\mathcal{M}(\mathcal{F}) = \{m \mid m \in \mathcal{I}(\mathcal{F}) \wedge \mathcal{D}(m)\}$$

is generated and added to the definitions of $\mathcal{C}_{\text{group}}$.

Let $\mathcal{M}^*(\mathcal{X})$ be the methods implemented by a class \mathcal{X} after these first translations. Now the methods are distributed as follows:

$$\mathcal{M}^*(\mathcal{C}) = \mathcal{M}(\mathcal{C}) \setminus \mathcal{G}(\mathcal{C})$$

$$\mathcal{M}^*(\mathcal{C}_{\text{group}}) = \mathcal{G}(\mathcal{C}) \cup \mathcal{M}(\mathcal{D})$$

5.3 Enable Inheritance among Synthetic Wrapper Classes

The second translation step related to `JAVAGROUPS` starts as soon as the first step from the original Java compiler has completed. This includes a detection if the elements that hold wrapper classes reside in an inheritance hierarchy, and if the elements superclasses include group methods, too. After that original compiler step, all information needed for an inheritance scan is present. This means that the information is accessible by the `ClassSymbols` that are generated during `ClassEnter` and stored in the appropriate classes. Using those `ClassSymbols` `JAVAGROUPS` scans for inheritance recursively, starting from the superclass of the element that contains a group method. If that superclass contains a suitable wrapper class, it stops the scan and lets the child's wrapper class inherit from that class. Otherwise `JAVAGROUPS` continues with the superclasses' superclass until either a wrapper class is found or the scan reaches the top of the inheritance hierarchy.

To illustrate this translation step, we need to extend our example and add some inheritance. We introduce a new class `Herring` (see [Listing 1](#)) that extends the `Fish` class.

```
public class Herring extends Fish {
    @Group
    public void swarmAttack(Creature creature) {
        super.swarmAttack(creature);
        this.tauntVictim();
    }
    @Group
    public void tauntVictim(){
        ...
    }
}
```

Listing 1. Introducing the Herring class

This class is subject to the same translation steps as its parent class. This means an inner class that is responsible for swarm behavior is generated and the methods are moved to the new class. Furthermore, as the wrapper class of `Herring` should extend the wrapper class of `Fish`, that inheritance information is added to the tree.

5.4 Transformation of Swarm Behavior Call Sites

As method resolution takes place during the Attribution phase, `JAVAGROUPS` uses it as a pointcut for adding the previously generated wrapper classes to the AST. If the original method lookup fails, we retry with the wrapper class of the call site, if such a class is present. If a group method is found and hence the method call is valid we have to wrap that call. This results in passing the original call site as parameter to the wrapper class and let that wrapper class receive the original call.

This step concludes the syntax tree translations done by `JAVAGROUPS`. The other passes of the compiler remain unchanged.

6 Discussion

This section discusses issues related to both Swarm Behavior in general as well as issues that are specific to the implementation of the `JAVAGROUPS` prototype.

— Currently, the only possible container type is `Collection`. This limits the interface available from within the group method. The implementation must be expanded to enable any container type. For example if the programmer has to use the `java.util.Queue` interface he cannot invoke `Queue` specific methods in the current implementation. To solve this issue it must be possible to define the class or interface that is used as container for each group method. The following

line of code suggests a possible implementation that passes the container type as parameter to the annotation.

```
@Group("java.util.Queue")
```

— Within the lexical scope of a group method defined in class `T` the pseudovariable `this` refers to a collection of `T` instances rather than to an instance of `T` as usual. This can lead to confusing situations when in the same file `this` refers to both an element instance in instance methods and to a container of elements in the group methods. For future releases of `JAVAGROUPS`, we thus plan to introduce a new pseudovariable `group` to address this issue.

— The synthetic wrapper classes use generics, this implies the limitation that from within a group method it is not possible to create new instances of the container's element type. This is a general limitation of generics in Java.

— In previous work we present a Swarm Behavior in Smalltalk [7]. Smalltalk is a dynamically typed language and thus the swarm behavior of a group is determined dynamically at runtime. The least common supertype of a collection changes as objects join or leave the group at runtime. This results in a problem with empty containers, as the least common supertype of an empty container's elements is undefined. For the Smalltalk implementation we have chosen to fail with an exception when group methods are sent to an empty collection. In Java we can do better. We use the generic type parameter of collection instances to determine the element's type. This implies that the swarm behavior of a group is not dynamic, as the least common supertype of its elements is declared at compile time. This solves the problem with empty collections mentioned above, since the generic type parameter is also known for empty instances.

7 Related Work

Swarm behavior is not the first approach to address group-related behavior. This section discusses related work on other group-related approaches.

There are also approaches such as *e.g.* C++ generics, that allow the programmer to express swarm behavior by seamlessly using the meta-programming facilities of the host languages. However, swarm behavior targets a way more specific problem than general-purpose meta-programming approaches. Swarm behavior extends the object model to decouple the behavior of collections from the collection hierarchy, and to extend collections with domain specific behavior that depends on the element type of a collection.

7.1 Array Programming and Higher Order Messages

Swarm behavior is not the same as array programming (*i.e.* projection of message sends). The principle behind array programming is that the same operation is

applied to an entire array of data, without the need for explicit loops. Apart from ancient languages such as APL, or mathematical software such as MathLab and Mathematica, array programming has been recently applied in the context of dynamic object-oriented languages by FScript [9], a Smalltalk-based scripting language for OSX, and by the ECMAScript for XML (E4X) specification, an extension of JavaScript.

Higher Order Messages (HOM) [10] can be seen as a generalization of array programming in the context of dynamic object-oriented languages. The basic operation in object-oriented languages is the message send. The fundamental principle behind HOM programming is that the same message is send to multiple objects. The principle of HOM programming is also known as cooperative call or message broadcast. The objects that a message is sent to can be those of an array or a collection, resulting in a form of array programming. In addition, HOM can be used on any composite structure of objects. It is not limited to arrays or collections.

Compared to swarm behavior, both array programming and higher order messages are too limiting to be useful in defining domain-specific behavior for collections of objects. Their concern is to manipulation groups of objects at once, but not to attach custom behavior to the group as a whole. Consider for example `showTagCloud()`, its semantics can not be achieved with messaging alone. Swarm behavior provides the required extension, as it allows the group's elements to attach custom behavior to the group.

7.2 Generics in Java and C#

This subsection compares swarm behavior to Java generics, as sometimes, programmers tend to confuse these two concepts. Even though both generics and swarm behavior are concerned with element types, they have not much in common. The concern of generics is type safety. A generic class `List<T>` is parameterized at compile-time, *e.g.* as `List<Book>`, to ensure type-safety of its elements. Generics have recently been introduced in C# and Java.

The concern of swarm behavior, on the other hand, is to extend a collection with domain-specific behavior based on the type of the collection's elements. Consider again `showTagCloud()`, that is defined in the class `Book` as a *group method*. Instead of having to write a specific `BookList` class, using swarm behavior, any collection who's elements are of type `Book` understands `showTagCloud()`, any collection of any collection type.

7.3 Predicate types

Swarm behavior may be considered a special kind of predicate classes [2]. Like a normal class, a predicate class has a superclass, methods and fields. However, unlike normal classes, any object may automatically become an instance of a predicate class whenever it satisfies a predicate condition associated with a predicate class. Considering again the library example, the swarm behavior of

`Book` is a predicate class `Book group` with the following predicate (given in OCL syntax)

```
context Book group pred:
self isKindOf Collection and
  self->forAll( each | each isKindOf Book )
```

7.4 Traits

Traits [4] are collections of methods (behavior) that can be composed into classes. Traits are normally seen as entities that are composed by the developer when designing the system. More dynamic notions of traits have been explored where traits are installed or retracted at runtime [1]. As traits are applied to classes, they provide behavior to all instances of the class, similar to normal methods. Thus traits alone do not help to model behavior of a collection of objects. It would be interesting to explore a combination of traits with swarm behavior. Traits could be used to structure the swarm behavior. As classes are composed of traits, groups could be composed from traits to support reuse and limit code-duplication.

7.5 Context Oriented Programming

ContextL [3, 6] is a language to support Context-Oriented Programming (COP). The language provides a notion of *layers*, which package context-dependent behavioural variations. In practice, the variations consist of method definitions, mixins and *before* and *after* specifications. Layers are dynamically enabled or disabled based on the current execution context. One can see swarm behavior as a form of context orientation: a collection has, in the context that all the elements are of a certain type, different behavior than in other contexts. Current COP languages do not provide a model for context that is powerful enough to express swarm behavior and therefore does not provide any help for modeling group behavior.

8 Conclusion

This paper presents *Swarm Behavior*, a new composition operator to associate behavior with a collection of instances. Swarm behavior extends the object model to decouple the behavior of collections from the collection hierarchy. Swarm behavior allows programmers to extend collections with domain specific behavior that depends on the element type of a collection.

Most current languages offer limited support for group-related behavior. We report and analyze three idioms that are commonly used to achieve group-related behavior. From this analysis we conclude a set of required properties for group-related behavior and propose Swarm Behavior as a solution.

- Swarm behavior is defined in the lexical scope of the element class.
- Swarm behavior is restricted to collections of the defining element type.
- Swarm behavior is not restricted to a specific subtype of collection.
- Swarm behavior parallels the inheritance hierarchy of the element types.

We provide SWARM-LOOKUP as an algorithm for swarm behavior lookup. SWARM-LOOKUP extend the normal lookup of collections. If no instance method definition is provided by the collection’s class, the most specific superclass of the collection elements is used as the base for a group lookup. Group lookup works in the same way as instance lookup, but operates on group methods rather than instance methods. Group methods are defined in the lexical scope of a class using the `@Group` annotation. Within a group method the pseudovariable `this` is bound to a collection of instances of the defining class.

We present the JAVAGROUPS prototype, an implementation of Swarm Behavior for Java. Our prototype implements Swarm Behavior by extending the Java compiler with a series of AST transformations. The prototype is open source and available online for download (see [Section A](#)).

As future work we would like to use the JAVAGROUPS compiler to refactor major Java applications towards Swarm Behavior. Previous work using a Smalltalk implementation of Swarm Behavior yielded promising preliminary results (upto 20% code reduction [7]) in this direction. Also we would like to further formalize the operational semantics of Swarm Behavior.

Acknowledgments

We thank Oscar Nierstrasz for his feedback and suggestions. We thank Toon Verwaest for his support during the first steps of the JAVAGROUPS project. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects “Bringing Models Closer to Code” (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010) and “Biologically inspired Languages for Eternal Systems” (SNF Project No. PBBEP2-125605, Apr. 2009 - Mar. 2010).

References

1. Alexandre Bergel and Stéphane Ducasse. Supporting unanticipated changes with Traits and Classboxes. In *Net.ObjectDays (NODE’05)*, pages 61–75, Erfurt, Germany, September 2005.
2. Craig Chambers. Predicate classes. In Oscar Nierstrasz, editor, *Proceedings ECOOP ’93*, volume 707 of *LNCS*, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.
3. Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) ’05, co-organized with OOPSLA’05*, pages 1–10, New York, NY, USA, October 2005. ACM.

4. Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.
5. David Erni. JAG — a prototype for collective behavior in Java. Bachelor’s thesis, University of Bern, August 2008.
6. Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), March 2008.
7. Adrian Kuhn. Collective behavior. In *Proceedings of 3rd ECOOP Workshop on Dynamic Languages and Applications (DYLA 2007)*, August 2007.
8. Doug Lea. Objects in groups. submitted ECOOP ’94SUNY at Oswego / NY Case Center, 1994.
9. Philippe Mougín and Stéphane Ducasse. OOPAL: Integrating array programming in object-oriented programming. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA ’03)*, pages 65–77, October 2003.
10. Marcel Weiher and Stéphane Ducasse. High-order messaging. In *Proceedings of International Symposium on Dynamic Languages (SDL’05)*, pages 23–34. ACM Press, 2005.

A JAVAGROUPS Quickstart

Our prototype implementation of swarm behavior, JAVAGROUPS, is available online at the following URL:

https://www.iam.unibe.ch/scg/svn_repos/Students/erni/bachelor/dist/jag.zip

The archive consists of 2 Files, `classes.jar` that contains the modified compiler classes and `jagc` that wraps the default javac command and adds the parameters necessary to load the modified compiler classes. Note that you will need at least Java 6 to be able to run JAVAGROUPS. Our modification enables Swarm Behavior by loading the modified Java compiler classes into the bootclasspath and hence overrides the default classes. This can be done by either using the above mentioned `jagc` shell script (for unix systems) that automates everything:

```
jagc <*.java>
```

or by adding the bootclasspath and classpath manually using the following command:

```
javac -J-Xbootclasspath/p:<path to classes.jar>
-cp <path to classes.jar> <*.java>
```

It is necessary to add `classes.jar` to the classpath because the annotation `@Group` is provided by that file. The annotation must be imported from `com.sun.tools.javac.group.Group`.

The following classes serve as example and show the basic functionality of JAVAGROUPS. The `Creature`, `Fish` and `Shark` classes model the running example of this paper. The `ExampleFight` class is the main class of the example that prints out the result to the command line.

These classes can now be compiled by typing the command

```
./jagc Creature.java Fish.java ExampleFight.java Shark.java
```

Compilation results in 5 class files, Creature.class, Fish.class, Shark.class, ExampleFight.class and Fish\$Group\$Fish.class that represents the inner class. The compiled program can now be started by typing

```
java ExampleFight
```

This prints the following on the command line:

```
Shark alive with 100 hitpoints.
Shark defeated with -9900 hitpoints.
```

The source code:

```
public class Creature {

    protected int hitpoints;

    public void damage(int damage) {
        this.hitpoints -= damage;
    }

    public boolean alive() {
        return hitpoints > 0;
    }
}

public class Shark extends Creature {

    public Shark() {
        this.hitpoints = 100;
    }

    public void attack(Creature creature) {
        creature.damage(10);
    }
}
```

```

import com.sun.tools.javac.group.Group;

public class Fish extends Creature {

    public Fish() {
        this.hitpoints = 1;
    }

    public void attack(Creature creature) {
        creature.damage( 0 );
    }

    @Group
    public void swarmAttack(Creature creature) {
        int swarmSize = this.size();
        creature.damage( swarmSize * swarmSize );
    }
}

import java.util.ArrayList;
import java.util.Collection;

public class ExampleFight {

    public static void main(String[] args) {
        // setup
        Shark shark = new Shark();
        Collection<Fish> swarm = new ArrayList<Fish>();
        for (int n = 0; n < 100; n++) swarm.add(new Fish());

        // example 1, aggregation
        for (Fish fish: swarm) fish.attack(shark);
        System.out.printf("Shark alive with %d hitpoints.\n",
            shark.hitpoints);
        // ==> 100

        // example 2, swarm behavior
        swarm.swarmAttack(shark);
        System.out.printf("Shark defeated with %d hitpoints.\n",
            shark.hitpoints);
        // ==> 100 - (10000) = -9900
    }
}

```