JAG - a Prototype for Collective Behavior in Java

David Erni
Supervision: Adrian Kuhn

University of Bern, March 2008

Abstract

Today's programming language models support a wide variety of mechanisms to share or define the behavior of objects but this often addresses a single object only. As there are operations that are related to a group of objects rather than to a single object, we propose a way to define how those objects operate as group, when stored inside a data container (e.g. Collection, Set, ...). We present a prototype of Collective Behavior in Java that enables some basic collective functionality.

Contents

1 Introduction 3

2 Example 4

3 Model of Collective Behavior 6

4 Collective Behavior in Java 10

5 Java Compiler in a Nutshell 11
- 5.1 Parse 12
- 5.2 Enter 13
- 5.3 Process Annotations 13
- 5.4 Attribute 13
- 5.5 Flow 14
- 5.6 Desugar 14
- 5.7 Generate 14

6 JavaGroups Javac Extension 14
- 6.1 Transformations during Enter 15
  - 6.1.1 Generate Group Classes 16
  - 6.1.2 Add Inheritance 18
- 6.2 Attribution 19

7 Discussion 19

8 Summary 20

Acknowledgments 20

References 21

List of Figures 21

Listings 21

A Getting Started 22

1 Introduction

Object oriented design revolves around the notion of objects and methods. An application is composed of a variable range of different objects. Often these objects are composed of groups that collaborate as a whole. Typically, a collection framework is available that provides containers for these groups. However, collections are abstract datatypes and hence do not offer content-specific methods on top of them. Worse, it is usually at the level of the language itself that such support is missing.

Consider for example Java. As in most object oriented languages, methods can be defined either related to instances or, in case of static methods, related to classes. However, we can not define content-specific methods that are related to multiple instances of the same type.

This thesis presents JavaGroups, an extension of the Java language, that allows programmers to defined content-specific methods on top of collections. We call this new language feature *Collective Behavior.* The key idea of Collective Behavior is to extend collections with content-specific behavior, i.e. behavior specific to their element's type. Collective Behavior is invoked on the collection instance, but defined as part of the element's class using the `group` keyword.

Figure 1 illustrates the collaborators of Collective Behavior.

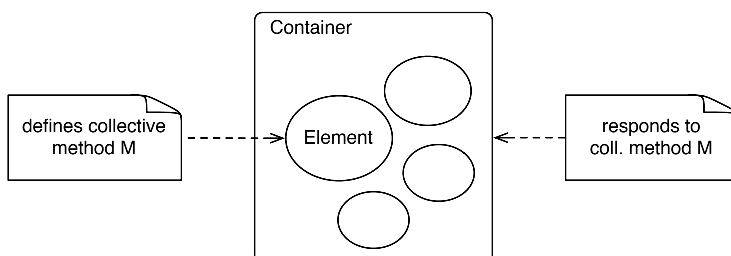

Figure 1: Container is an instance of `Collection` that contains elements of type `T`. The collective method `M` is defined on the type `T` and can be invoked on the collection instance.

This thesis uses the following terminology to denote the collaborators of Collective Behavior.

Element — an instance of type `T` enclosed by a container.

Container — an instance of `Collection` that holds elements of type `T`.

Collective method — a method `M` that can be invoked on any container that holds elements of type `T`. The method `M` is defined as part of the element's type `T` rather than on the collection's type.

The remainder of this thesis is structured as follows. Section 2 discusses current options to define operations related to a group of objects in Java. Section 3 defines the model of Collective Behavior. Section 4 discusses in detail how JavaGroups extends the object model of Java with Collective Behavior. Section 5 summarizes the OpenJDK Java compiler passes and their documentation respectively. Section 6 presents how we extended the Java compiler with JavaGroups. Section 7 discusses the open issues. And finally, Section 8 summarizes.

2 Example

This section discusses current options to define operations related to a group of objects in Java, and eventually presents how JavaGroups extends the Java syntax with collective method definitions.

Given the small example class in Listing 1, we define a class `Vehicle` with a `price` property. The remainder of this section elaborates how to define a method that computes the average price of a group of vehicles.

Listing 1: Example vehicle class

```java
public class Vehicle {

    protected float price;

    public Vehicle(float price) {
        this.price = price;
    }

    public float getPrice () {
        return this.price;
    }
}
```

Assume that we want to define behavior related to a group of objects. One option is to define a static helper method (see Listing 2) that resides somewhere in the project. This is clearly a workaround, as static methods break the object oriented metaphor. For example, static methods do not support polymorphism.

Listing 2: Implementation using static helper methods

```java
public static float averagePrice(Collection<Vehicle>
    container) {

    float result = 0;
    for(Vehicle vehicle : container){
        result += vehicle.getPrice();
    }
    return result / container.size();
}
```

Another option is to extend the container type with a custom subclass, and then define the desired behavior in the subclass. This is problematic because there are multiple collection classes. Hence we have to extend all of them to provide the collective method. This does not scale because there are n possible collection classes and m possible element classes.

Listing 3 illustrates this option. The class `VehicleList` extends a collection class, `ArrayList` in this case, and adds a content specific method `averagePrice`. The obvious disadvantage here is that the added behavior is only available for Vehicles that are stored in an ArrayList. Furthermore if the implementation has to use different Collection classes (e.g. List, Set, ...) for some reason, it has to extend every class and add the Collective Behavior as it is not possible to inject new functionality into an existing inheritance hierarchy.

Listing 3: Implementation by extending the container

```java
public class VehicleList extends ArrayList<Vehicle> {

    public float averagePrice() {
        float result = 0;
        for (Vehicle vehicle : this) {
            result += vehicle.getPrice();
        }
        return result / this.size();
    }
}
```

In languages that allow us to extend core classes, there is a third option. We could extend the core classes and add Collective Behavior directly by monkey patching. The problem of this option is, that the collective methods defined on the core classes are made available for collections holding any type of element, and not just on containers holding elements of type T. Hence the core classes are bloated unnecessarily by this workaround.

Our implementation suggests a new way to define the behavior related to a group of objects. It enables calling collective methods directly on the container as illustrated in Listing 4. The method definition, however, is done

in the lexical scope of the element. This implies, that collective methods must be labeled in some way as they need to differ from the common method definitions. In our implementation, we chose the `@Group` Annotation to label collective methods. Listing 5 shows the definition of the collective method `averagePrice` on the `Vehicle` class.

Please note that inside the collective method's body the `this` keyword is bound to the container.

Listing 4: Calling a collective method on a container

```
    Collection<Vehicle> container = new ArrayList<Vehicle>();
    ...
    float average = container.averagePrice();
```

Listing 5: Defining a collective method on an element

```
public class Vehicle {

    @Group
    public float averagePrice(){
        float result = 0;
        for (Vehicle vehicle : this) {
            result += vehicle.getPrice();
        }
        return result / this.size();
    }
}
```

3 Model of Collective Behavior

Collective Behavior extends the common object model and adds a new option to organize methods. It makes it possible to model the behavior of a group of elements (that are stored in a container) in the element's type rather than in the container's type. This is achieved by extending the object model with a new kind of method. In addition to instance and static methods, we allow classes to define collective methods.

Collective methods are defined as part of the element's type `T`, and invokable on any container that contains elements of type `T`. However, collective methods are not invokable on single instances of type `T`!

Which collective methods are invokable on a container depends on the types of the elements stored in that container.
Kuhn defines Collective Behavior as follows [4]:

> Collective behavior provides a way to associate behavior with multiple instances of the same class. Collective behavior extends

> the object model to solve a structuring problem that is not addressed by common class- and object-based concepts and notations.

He elaborates further:

> The idea of collective behavior is to define element-specific methods of collections in the model of the elements class rather than in the collections class. Collective behavior extends the common object-model with a new kind of class-method association, attaching the collective methods to classes.

Kuhn presents an implementation for Smalltalk, a dynamically typed language. The collective behavior of a group is determined dynamically at runtime. The least common supertype of a container's elements changes whenever objects join or leave the group. This results in a problem with empty containers, as the least common supertype of an empty container's elements is undefined.

In Java we can use the generic type parameter of collection instances to determine the element's type. This implies that the collective behavior of a group is not fully 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.

Figure 2 illustrates how JavaGroups models collective behavior in Java by giving a concrete example. It shows the availability of methods and distinguishes between the `Vehicle` class, the `Container` class, and instances thereof. The important difference to the common Java model is, that instances of `Container` understand collective methods of `Vehicle`. As `Container` is parameterized with `Vehicle` it may only contain objects that are an instance of `Vehicle`. The collective methods that are available on the given `Container` are actually defined in `Vehicle`'s class.

Moreover, the collective hierarchy may also include inheritance. Given a class `Car` that extends the `Vehicle` class, collective methods that are available for a `Collection` of `Vehicles` must also be available for Cars, as every `Car` is-a `Vehicle`.

These changes imply some modifications for the method lookup on the containers as it has to take into account collective methods as well as methods that were defined the common way. The following list describes the changed lookup:

1. Start Java's common lookup on the container type. This implies that collective methods cannot override methods that are already defined on the Container class.

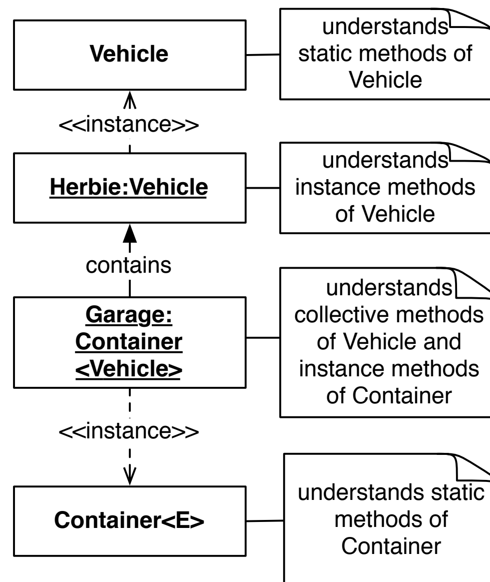

Figure 2: Collective Methods are defined on Vehicle but invokable on a Collection that contains instances of Vehicle. The common Method's definition and availability does not change. Note that collective methods can't be called on an instance of Vehicle.

2. Let T be the type parameter (`Container<T>`). If above lookup fails, start lookup for collective methods in T.

3. If lookup at T fails, continue with T's supertype until Object is reached (and there is no supertype any more). If no collective method is found, fail with an error.

Figure 3 illustrates a sequence diagram of collective method lookup on an instance of `Container`. Assume we call a method on a `Collection` of `Cars`. The selected method is a collective method defined for a group of `Vehicles`, and hence it is available for a group of `Cars`, too. First, we check if the method is a method available for instances of `Collection`. This is not the case, so start the lookup for collective methods in `Car` as the container (in this case `Collection`) is instantiated with the type parameter `Car`. If the collective method is not found for `Car`, continue with `Car's` supertype, in this case `Vehicle`. In that example, the collective method is found in `Vehicle's` type and hence 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.

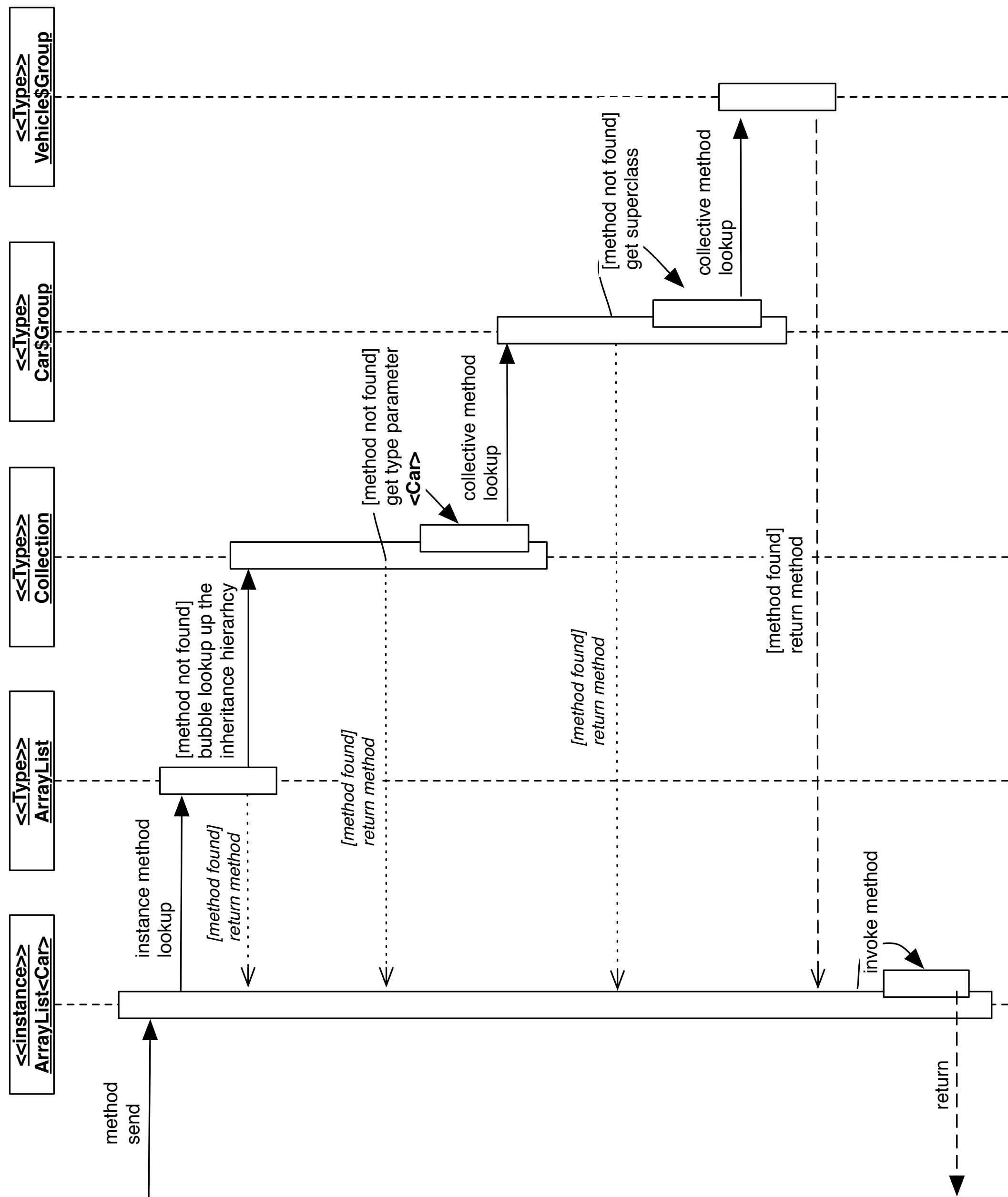

Figure 3: Example method invocation on a Collection of Cars. Let the selected method be a collective method defined on Vehicle. After the common lookup failed we start the lookup for collective methods in Car. Finally the collective method is found on Vehicle, returned, and invoked. The dotted lines indicate where, in a general case, the method can be found. The dashed lines indicate where the lookup stops in this concrete example.

4 Collective Behavior in Java

This section discusses in detail how JavaGroups extends the object model of Java with Collective Behavior.

Java offers no possibilities for monkey patching, therefore we introduce a wrapper class.

Our idea is to introduce a wrapper class that wraps invocations of collective methods. The call site is changed from the container (that does not implement collective methods) to a helper class that implements them. However, this helper class needs to know about the elements stored in the original container, hence the helper class has to get a reference to that container. This is solved by passing the container as constructor argument.

As a collective method is logically bound to the container, the keyword `this` within a collective method's body refers to the container. This implies that, from the collective method's body, a container method must be invokable, too. Therefore, the wrapper class must delegate calls to the container's methods to the instance of the container that is passed in as constructor argument. For example a call to `container.doCollectiveStuff()` on a container that is parameterized with `Car` is wrapped into `new Car.Car$Group(container).doCollectiveStuff()`.

Such a wrapper class can handle the Collective Behavior of exactly one Element. For every Element containing collective methods, a wrapper class must be generated. We will refer to that wrapper class as Group class for an element.

Group class — Group classes are responsible to handle the Collective Behavior for one element. A Group Class implements collective methods of its corresponding Element. It is used as a helper class and wraps invocations of collective methods.

In our reference implementation, each class that defines collective methods is transformed as follows:

1. Let T be a class that implements collective methods. First we generate a group class for T, which is responsible to handle T's collective behavior.

2. If one of T's superclasses defines collective methods, add inheritance information to the group class generated in the first step. This means, let T's group class inherit from the group class of the closest class up T's inheritance chain, that implements collective methods.

3. Because the original container is a generic class with one type parameter, we know the upper bound of the elements stored in that container. This type parameter is then used to parameterize the Group class itself. This parameterization is needed to be able to call methods defined

on the elements from within the collective method's body. This implies the limitation that it's not possible to create new instances of the elements stored in the containers [2].

4. The group classes are then parameterized by passing the original container as a constructor parameter. This happens by scanning the files for invocations of these collective methods and wrapping the receiving collection into a new instance of the group class. Please refer to Section 6 for more details.

Figure 4 illustrates these steps. It shows two example classes `Vehicle` and `Car`, that extends `Vehicle`. For both classes a, group class is generated and added as inner class. The group class is then parametrized with its corresponding element T. The type parameter of the group class is set to `E extends T`. `Car's` group class extends `Vehicle`'s group class because `Car` extends `Vehicle`, and both of them define collective methods. Both group classes delegate to `Collections` that contain instances of their corresponding elements.

For a formal definition and further details of the implementation of these steps in Java, please refer to Section 6.

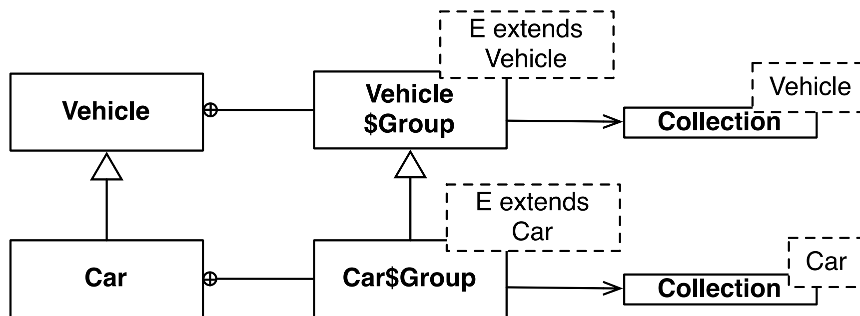

Figure 4: Given two example classes `Vehicle` and `Car` extending `Vehicle`. For both classes a group class is generated and added as inner class and then parametrized with its corresponding element. `Car`'s group class extends `Vehicle`'s group class just like `Car` extends `Vehicle`. Both group classes delegate to a parametrized `Collection`.

5 Java Compiler in a Nutshell

This chapter summarizes the OpenJDKJava compiler passes and their documentation respectively [5]. To explain our Java compiler modifications, we need to get a brief overview over the compiler first. This chapter contains a short introduction of the main compiler passes.

The Java compiler is open source. It is available as part of the OpenJDK[1] project, and can be downloaded form the java compiler group website[2].

The compilation process is managed by the Java Compiler class that is defined in `com.sun.tools.javac.main`. Figure 5 illustrates the passes that are performed when the java compiler is invoked with default compile policy.

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 specifed 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.

Figure 5: When the Java compiler is invoked with default compile policy it performs these passes: Parse, Enter, Annotate, Attribute, Flow, Desugar, and Generate.

5.1 Parse

To parse the files, the compiler relies on the classes available under `com.sun.tools.javac.parser`. As a first step, the lexical analyzer maps an input stream consisting of characters into a token sequence. Then the parser maps the generated token sequence into an abstract syntax tree.

[1] http://www.openjdk.org/

[2] http://www.openjdk.org/groups/compiler/

5.2 Enter

During this pass the compiler registers symbols for all the definitions that were found into their enclosing scope. Enter consists of the following two phases:

During the first phase, all class symbols are registered within their corresponding scope. This is implemented by using a visitor that descends down the hierarchy, visiting all classes, including inner classes. The compiler attaches a `MemberEnter` object to each class symbol that will be used for completion in phase 2.

In the second phase these classes are completed using the `MemberEnter` object mentioned above. First step during completion is the determination of a class's parameters, supertype and interfaces. In a second step those symbols are entered into the class's scope (ignoring the class symbols that have been entered during the first step). Unlike the first phase, the second one is lazily executed. The members of a class are entered as soon as the contents of a class are first accessed. This is achieved by installing a completer object in the class symbols. Those objects can invoke the member-enter phase on-demand.

Finally, enter adds all top-level classes to a todo-queue that will be used during the Attribution pass.

5.3 Process Annotations

If an annotation processor is present and annotation processing is requested this pass processes any annotations found in the specified compilation units. JSR 269 defined an interface for writing such plugins [3]. However, this interface is very limited and does, in particular, not allow one to extend the language with Collective Behavior. The main limitation being that JSR 269 does not provide sub-method reflection.

However, Kuhn and Erni show, that it is possible to implement an annotation processor that does AST rewriting by casting JSR objects to their compiler internal interfaces [1].

5.4 Attribute

Attributes all parse trees that are found in the todo-queue generated by the Enter pass. Note that attributing classes may cause additional files to be parsed and entered via the SourceCompleter.

Most of the context-dependent analysis happens during this pass. This includes name resolution, type checking and constant folding as subtasks. The main subtasks are performed by the following auxiliary classes.

- **Check:** Does type checking. It will report errors such as completion errors or type errors.

- **Resolve:** Does name resolution. It will report errors if the resolution fails.
- **ConstFold:** Does constant folding.
- **Infer:** Does type parameter inference.

5.5 Flow

This pass performs data flow checks on the previously attributed parse trees. Liveness analysis checks that every statement is reachable. Exception analysis ensures that every checked exception that is thrown is declared or caught. Definite assignment analysis ensures that each variable is assigned when used. Definite unassignment analysis ensures that no final variable is assigned more than once.

5.6 Desugar

Removes syntactic sugar, such as inner classes, class literals, assertions, and foreach loops.

5.7 Generate

This (final) pass generates the source or class file for a list of classes. It decides to generate a source file or a class file depending on the compiler's options.

6 JavaGroups Javac Extension

This section presents how we extended the Java compiler with JavaGroups. We show which steps of the compiler are extended, it is the Enter and the Attribute step.

To implement Collective Behavior in the Java compiler, several things need to be done. First of all, we need a way to tag certain methods as collective methods. In JavaGroups this is achieved by adding the annotation **@Group** to those methods, see Listing 6.

Listing 6: Label a method as collective method

```
@Group
public float averagePrice() {
    ...
}
```

Figure 6 illustrates the two major modifications that need to be done during compilation:

1. a) Generate group classes. This means scan for collective methods and generate group classes for all toplevel classes that contain collective methods. Furthermore, the delegation interface for the collective classes needs to be generated. Currently, the container type defaults, and is limited to `java.util.Collection`. This means that by using the `this` keyword you are able to call any `Collection` method from the collective method's body, but no one that is defined further down the collection inheritance hierarchy.

   b) Add inheritance to the generated group classes. This includes a detection if the elements that define collective classes reside in an inheritance hierarchy, and if the elements' superclasses define collective methods, too. This detection happens by scanning for inheritance recursively, starting from the superclass of the element that defines a collective method. If that superclass defines suitable group class, stop and inherit from that class. Otherwise continue with the superclasses superclass until either a group class is found or the scan reaches the top of the inheritance hierarchy.

2. Change the calls sites of collective methods. We wrap every method invocation that calls a collective method with the group class that is generated in the first step.

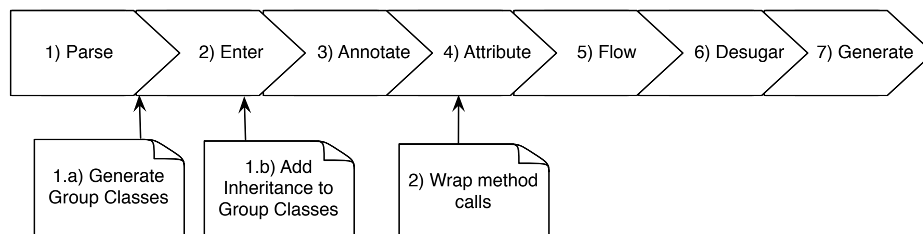

Figure 6: Before the Compiler starts the Enter pass, JavaGroups generates the group classes. Later on during enter, JavaGroups adds inheritance information to the previously generated group classes. During attribution, JavaGroups wraps the method calls to collective methods into the group classes.

6.1 Transformations during Enter

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

6.1.1 Generate Group 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, JAG performs the following transformations for each class $\mathcal{C}$.

- **add an inner class:** For each class $\mathcal{C}$ where $\mathcal{G}(\mathcal{C}) \neq \emptyset$ a group 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<$\mathcal{C}$> as parameter. This parameter is later used to refer to the original collection for delegation.

- **moving collective 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 container methods:** The final step for this pass is to generate the methods needed for delegation. Let $\mathcal{F}$ be the container's 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

  $$\mathcal{D}(m) = \{\text{m delegates to the Collection passed to } \mathcal{K}\}$$
  $$\mathcal{M}(\mathcal{F}) = \{m \mid m \in \mathcal{I}(\mathcal{F}) \wedge \mathcal{D}(m)\}$$

  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:

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

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

In Listing 7 we extend the example from the previous chapter (Listing 1) and add a collective method. To explain what we do internally, this will serve as the running example for the following descriptions.

Listing 7: Collective method before the first compiler translation

```
public class Vehicle {

    @Group
    public float averagePrice() {
        float result = 0;
        for (Vehicle vehicle : this) {
            result += vehicle.getPrice();
        }
        return result / this.size();
    }

}
```

After the first translation steps during enter, the class will look like as given in Listing 8.

Listing 8: Collective method after the first compiler translation

```
public class Vehicle {

    public static class Group$Vehicle<E extends Vehicle> implements
         Collection<E> {

        protected Collection<E> delegate;

        public MyElement$Group(Collection<E> delegate){
            this.delegate = delegate;
        }

        @Group
        public float averagePrice() {
            float result = 0;
            for (E vehicle : this) {
                result += vehicle.getPrice();
            }
            return result / this.size();
        }

        public boolean add(E param0){
            return this.delegate.add(param0);
        }

        // other collection methods follow
        ...
```

```
    }
}
```

6.1.2 Add Inheritance

The second translation step related to JavaGroups starts as soon as the first step from the original Java compiler, as mentioned in Subsection 5.2 has completed. This includes a detection if the elements that hold collective classes reside in an inheritance hierarchy, and if the element's superclasses include collective 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 collective method. If that superclass contains a suitable group class, it stops the scan and lets the child's group class inherit from that class. Otherwise JavaGroups continues with the superclasses' superclass until either a group 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 `Car` (see Listing 9) that extends the `Vehicle` class.

Listing 9: Introducing the car class

```
public class Car extends Vehicle {

    @Group
    public float averagePrice() {
        float avg = super.averagePrice();
        return this.doSomethingElse(avg);
        }

    @Group
    private float doSomethingElse(float f){
        ....
    }
}
```

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

6.2 Attribution

As method resolution takes place during the Attribution phase, JAG 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 group class of the call site, if such a class is present. If a collective method is found and hence the method call is valid we have to wrap that call in the group class. This results in passing the original call site as parameter to the group class and let the collective class receive the original call. Listing 10 shows an example of such a call before translation. Listing 11 represents the same code after translation.

Listing 10: Call to collective method before translation

```
Collection<Vehicle> container = new ArrayList<Vehicle>();
...
container.averagePrice();
```

Listing 11: Call to collective method after translation

```
Collection<Vehicle> container = new ArrayList<Vehicle>();
...
new Vehicle.Group$Vehicle(container).averagePrice();
```

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

7 Discussion

As JavaGroups is a prototype, there are some open issues.

- Currently, the only possible container type is `Collection`. This limits the interface available from within the collective 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 collective method. Listing 12 suggests a possible implementation that passes the container type as parameter to the annotation.

  Listing 12: Passing the container type as annotation parameter

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

- The lookup of methods is changed by adding a helper class that wraps method calls. For a proper implementation, the lookup must be modeled as part of the language's model. Such changes in the method

lookup imply changes in the way these calls compile to byte-code or even in the virtual machine itself.

- It is not possible to add collective methods to classes if the source-code is not available. This is because all modifications happen during compile time and are based on source-code analysis.

8 Summary

We presented a new way to solve a structuring problem that is not covered by current concepts. We introduced the notion of elements, containers and collective methods. The current implementation enables the definition of collective behavior in the element's domain. These collective methods are defined in the elements class but called on an instance of a container.

We presented a summary of the Java compiler and gave a short introduction to the main compiler passes. Eventually we presented a prototype that uses the Java compiler to implement Collective Behavior in Java.

Acknowledgments

I would like to thank Toon Verwaest for his support during the first steps of this project.

References

[1] David Erni and Adrian Kuhn. The Hackers Guide to Javac. March 2008.

[2] Gilad Bracha. The Java Tutorials: Generics. http://java.sun.com/docs/books/tutorial/extra/TOC.html.

[3] Joseph D. Darcy. JSR-000269 Pluggable Annotation Processing API, December 2006. http://jcp.org/en/jsr/detail?id=269.

[4] Adrian Kuhn. Collective behavior. In *Proceedings of 3rd ECOOP Workshop on Dynamic Languages and Applications (DYLA 2007)*, August 2007.

[5] The Java programming-language compiler (javac) group. http://openjdk.java.net/groups/compiler/.

List of Figures

1 Elements, Containers and Collective Methods 3
2 Model of Collective Methods 8
3 Example Lookup for Collective Methods 9
4 Generating Group Classes as Inner Class including Inheritance and Delegation 11
5 Unmodified Java Compiler Passes 12
6 Modified Java Compiler passes 15

Listings

1 Example vehicle class 4
2 Implementation using static helper methods 5
3 Implementation by extending the container 5
4 Calling a collective method on a container 6
5 Defining a collective method on an element 6
6 Label a method as collective method 14
7 Collective method before the first compiler translation 17
8 Collective method after the first compiler translation 17
9 Introducing the car class 18
10 Call to collective method before translation 19
11 Call to collective method after translation 19
12 Passing the container type as annotation parameter 19
13 Example Person class 23
14 Example Call class 23

A Getting Started

Jag can be downloaded from

```
https://www.iam.unibe.ch/scg/svn_repos/erni/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 Java6 to be able to run JavaGroups.

Our modification enables Collective Behavior by loading the changed 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 already provided by that file. To annotate collective methods with the `@Group` annotation, the annotation class can be imported from `com.sun.tools.javac.group.Group`.
The following 2 classes serve as example and show the basic functionality of JavaGroups. The Person class, given in Listing 13, represents a Person that has one attribute income and a collective method `avgIncome` that calculates the average income for a group of persons. The Call class as given in Listing 14 serves as a runner class. It populates a `Collection` with some `Persons`, invokes the collective method on that `Collection`, and finally prints out the result to the command line.

These 2 classes can now be compiled by typing the command

```
./jagc Person.java Call.java
```

Compilation results in 3 class files, Person.class, Call.class and Person$Group$Person.class that represents the inner class. The compiled program can now be started by typing

```
java Call
```

This prints the following on the command line:

```
The result is: 6.0
```

Listing 13: Example Person class

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

public class Person {

    private int income;

    public Person(int inc){
        this.income = inc;
    }

    public int getIncome() {
        return this.income;
    }

    @Group
    public float avgIncome() {
        float result = 0;
        for (Person p : this){
            result += p.getIncome();
        }
        return result/this.size();
    }

}
```

Listing 14: Example Call class

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

public class Call {

    public static void main(String[] args) {
        Collection<Person> myCol = new ArrayList<Person>();
        myCol.add(new Person(3));
        myCol.add(new Person(6));
        myCol.add(new Person(9));
        float result = myCol.avgIncome();
        System.out.println("The result is: " + result);
    }
}
```