Collective Behavior

Adrian Kuhn

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

Abstract. When modelling a system, often there are properties and operations related to a group of objects rather than to a single object only. For example, given a person object with an income property, the average income applies to a group of persons as a whole rather than to a single person. In this paper we propose to extend programming languages with the notion of collective behavior. Collective behavior associates custom behavior with collection instances, based on the type of its elements. However, collective behavior is modeled as part of the element's rather than the collection's class. We present a proof-of-concept implementation of collective behavior using Smalltalk, and validate the usefulness of collective behavior considering a real-life case study: 20% of the case-study's domain logic is subject to collective behavior.

1 Introduction

While object-oriented programming offers first-class concepts to model the behavior of single objects, the same is not true for groups of objects. 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 common class- and object-related concepts and notations. The aim of this paper is to provide a new treatment for the behavior of a collection of objects, in particular, for behavior that is based on the element's type rather than the container's type.

Briefly, the idea of collective behavior is that element-specific methods of collections shall be defined in the element's class. The concept of collective behavior works as follows:

Collective methods implement behavior specific to a group of objects (defined as a group by being member of the same collection). Collective methods are executed in the context of the container, using late binding based on the least common supertype of the contained members. Hence, even though Collective method are invoked on the group's container, they are modeled as part of the element's type.

A group's collective behavior is dynamic. As objects join and leave a group, the collective behavior, which depends on the least common supertype of the collection members, may change dynamically at runtime. The concept of a group-related behavior is by no means novel. The present treatment differs from others

mainly in attempting to integrate group-related behavior more firmly when modelling object-oriented system.

The remainder of this paper proceeds as follows: Section 2 exercises an introductionary example. Section 3 compares collective behavior to other group-specific object-oriented constructs. Section 4 discusses the object-model of collective behavior. Section 5 presents an implementation of collective behavior in Smalltalk. Section 6 refactors a large application to validate collective behavior. And, eventually, Section 7 concludes.

2 Example

Often an example is worth a thousand words. Hence, before proceeding, we give a short example, defining Collective method for `sum` and `average` on groups of `Integer` instances[1]. The examples are given in Smalltalk syntax.

```
Integer group >> sum
    ^self inject: 0 into: [ :a :b | a + b ]

Integer group >> average
    ^self sum / self size
```

Above the definition of `sum` and `average`. The methods are defined on `Integer group` rather than `Integer` (as would be the case for instance methods) or `Integer class` (as would be the case for static methods). Thus, they are Collective method—defined as part of `Integer`'s model and executable on any collection consisting of integer elements.

Within a collective methods `self` (which is how `this` is called in Smalltalk) refers to the group's collection. In the examples given above, the type of `self` is both `Collection` and `Integer group`. Hence, the methods `inject:into:` and `size` are looked-up in `Collection`, whereas `average`'s call to `sum` is looked-up in `Integer group`.

In the following, the `sum` message is sent to 1) a single integer instance, 2) a collection of integer, 3) a collection of floats, and 4) a mixed collection of integers and floats.

```
10 sum => MessageNotUnderstood exception.
```

The single integer instance above fails to lookup `sum`, as it is defined on `Integer group` rather than `Integer` itself. Collective methods can not be invoked on single instances of their defining type.

```
#(1 2 3 4) sum => 10.
```

The integer collection above successfully looks up `sum` in `Integer group`, and executes it as if being one of its own methods. That is, `sum` is executed once on the whole collection with `self` referring to the collection.

[1] Please note, to better exercise the example that we define `sum` and `average` on `Integer` rather then on the superclass `Number`.

```
#(1.0 2.0 3.0 4.0) sum => MessageNotUnderstood exception.
```

The float collection above fails to lookup `sum`, as `Float group` does neither define `sum` nor is a subgroup of `Integer group` (in Smalltalk, both `Float` and `Integer` subclass `Number`). The inheritance hierarchy of collective behavior parallels the hierarchy of the defining types, *i.e.* `Integer group` subgroups `Number group` which in turn subgroups `Object group`.

```
#(1 2 3 4.0) sum => MessageNotUnderstood exception.
```

The mixed collection above also fails to lookup `sum`, as its collective type is `Number group` which does not define `sum`. The type of a group is resolved using least common supertype, hence the above collection containing both `Integer` and `Float` elements has collective type `Number group`.

3 Related Constructs

The concept of collective behavior fills a gap in the analysis, design, and implementation of object systems. 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.

For example, given a library application with books and tags, the method `showTagCloud()` is associated as *collective methods* with the `Book` class. At runtime `showTagCloud()` may be invoked on any collection whose element's least common supertype is `Book`. Even collections created and returned by third-party code understand the additional behavior.

In the absence of collective behavior however, the method `showTagCloud()` has to be implement, either as a static helper method `showTagCloud(books)`, which takes the collection as argument, or using a helper class `BookList`. In any case, collections created and returned by third-party code require special handling as they do not understand `showTagCloud()`.

Associating group-related behavior with the class of a collection's elements rather then the collection class, avoids workarounds as the above helper method and/or class.

3.1 Array Programming

Collective behavior is not the same as array programming. The fundamental principal behind array programming is that the same message is send to an entire collection of objects, without the need for explicit loops. The principle of array programming is also known as cooperative call or message broadcast. 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[4], a Smalltalk-based scripting

language for OSX, and by the ECMAScript for XML (E4X) specification, an extension of JavaScript.

However, array programming is too limiting to be useful in defining domain-specific behavior for collections of objects. Array programming's 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 array programming alone. Collective behavior provides the required extension, as it allows the group's elements to attach custom behavior to the group.

3.2 Generics

This subsection compares collective behavior to generics, as sometimes, people tend to confuse these two concepts. Even though both generics and collective 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 collective behavior, on the other hand, is to adapt a collection's behavior to the type of its elements. Depending on the elements's type, a collection may exhibits different element-specific behavior. Consider again `showTagCloud()`, that is defined in the class `Book` as a *collective methods* . Instead of having to write a specific `BookList` class, using collective behavior, any collection who's elements are of type `Book` understands `showTagCloud()`, any collection of any collection type.

3.3 Predicate types

Collective behavior may be considered a special kind of predicate classes[1]. 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 collective 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 )
```

But unlike predicate classes, which are a general language extension, collective behavior is focused on the notion of executing a method "on" a group of objects rather than a single instance. Collective behavior aims to fix the semantics of classes and collections in object-oriented modelling. It is in order to model group-related behavior as a first-class association, that collective behavior extends the language.

4 Model

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

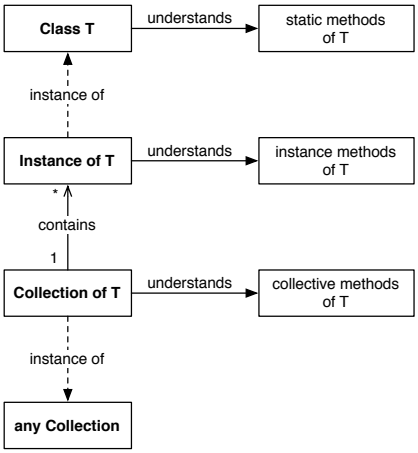

Fig. 1. Collective behavior is defined as part of `T`'s model and invokable on any collection with `T` elements. Collective methods can not be invoked on single instance of type `T` or on its class.

Figure 1 illustrates the object-model of class `T` involving collective behavior. Collective methods are defined as part of `T`'s model and invokable on any collection with `T` elements. Collective methods can not be invoked on single instance of type `T`, neither on its class.

Collective 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 element-specific behavior as well as the collection's class hierarchy. Method lookup of selector f on collection C is extended as follows

1. If a common lookup, starting at C's class has not been successful,
2. let T_0 be the least common supertype of C's elements.
3. Continue the lookup of f with the collective behavior of $T = T_0$,
4. if not found, proceed with the collective behavior of T's superclass,
5. repeat the previous step, until and including $T =$ `Object`.
6. If not successful so far, fail with `MessageNotUnderstood` exception.

As Collective method are called on collections rather than instances of `T`, the semantics of `this` are differ between instance and Collective method. Within

the scope of a collective methods, `this` refers to the targeted collection instance and is subject to the above lookup change.

4.1 Collective Behavior of Empty Collections

An empty collection has no elements, the least common supertype of its elements is undefined. Thus, the element-specific behavior of empty collection is undefined. This is an open issue, and we can not offer a satisfying solution yet. In the current reference implementation, calls to Collective method on empty collection raise a `MessageNotUnderstood` exception.

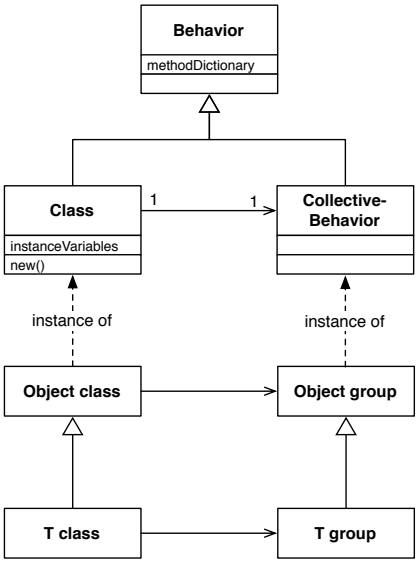

Fig. 2. Meta-model of collective behavior's reference implementation. Each instance of `Class` (modelling a class's instance behavior) has an associated instance of `CollectiveBehavior` (modelling the class's collective behavior).

5 Implementation

First and foremost, collective behavior extends the object *model* to solve a structuring problem. At the technical level of language constructs, collective behavior may be implemented using different approach. For example, an implementation of collective behavior may use aspect-oriented mechanisms, or collective behavior may be implemented using behavioral reflection[5]. The reference implementation present here is done using Smalltalk, and intercepting `Collection`'s `doesNotUnderstand:` to change method lookup[2].

Figure 2 illustrates the meta-model of collective behavior's reference implementation. Collective behavior extends Smalltalks's system class `Behavior`, and adds an association between `Class` and `CollectiveBehavior` to navigate from a class to its collective behavior.

```
Class >> group
    ^CollectiveBehavior forClass: self

CollectiveBehavior class >> forClass: aClass
    ^GlobalDictionary at: aClass ifAbsentPut: [
        | collectiveBehavior |
        collectiveBehavior := self new.
        collectiveBehavior elementClass: aClass.
        collectiveBehavior methodDictionary: MethodDictionary new.
        aClass superclass ~~ nil ifTrue: [
            collectiveBehavior superclass: (aClass superclass group) ].
        collectiveBehavior setInstanceFormat: (Behavior
            formatFromType: Collection behaviorType
            super: Collection
            instVars: '').
        collectiveBehavior ]
```

The lookup of methods on collection instances is intercepted in `Collection`'s `doesNotUnderstand:` method, which is called whenever a lookup a collection instance fails.

```
Collection >> doesNotUnderstand: aMessage
    ^self group
        performMessage: aMessage using: self
        ifDNU: [ super doesNotUnderstand: aMessage ]

CollectiveBehavior >> performMessage: aMessage using: aCollection ifDNU: dnuBlock
    ^(self canUnderstand: aMessage selector)
        ifTrue: [ (self lookupCompiledMethodAt: aMessage selector)
            valueWithReceiver: aCollection
            arguments: aMessage arguments. ]
        ifFalse: [ dnuBlock value ]
```

Then lookup itself is implemented in `performMessage:using:ifDNU:`, that uses the method dictionary and lookup as inherited from `Behavior` to lookup the collective methods, thus using the same lookup semantics for Collective method as for instance methods. If a matching method is found, it is evaluated using the passed collection `aCollection` as receiver.

6 Validation

To validate collective behavior's advantage over common object-oriented modelling concepts, we analyzed Moose[2], an object-oriented sized application in-

[2] The author of this paper is a member of Moose's development team. Please refer to http://smallwiki.unibe.ch/moose for more information on Moose.

cluding 827 classes and over 10'000 methods, and refactored it towards collective behavior. Moose is a software analysis platform, featuring a meta-model driven kernel and over a dozen of plug-ins, of which, six have been included in the case-study.

6.1 Analysis of Moose

Analysis of the system identified 25 classes subject to collective behavior. All 25 classes are helper class, implementing collective behavior for one related domain class. Table 1 lists these findings: the 25 helper classes feature a total of 197 Collective method, containing almost 20% of the system's domain logic.

Moose	– total	– affected	– percent	– domain	– affected	– percent
Classes	827	25	3.02%	123	25	20.3%
Methods	10'503	197	1.87%	2111	197	9.3%
Statements	35'149	1386	3.94%	6721	1386	20.6%

Table 1. Analysis of collective behavior in Moose. Almost 20% of the system domain logic, 1386 out of 6721 SLOC, are subject to collective behavior.

All helper classes follow the naming scheme: `<T>Group` related to `FAMIX<T>`. For example, helper class `Moose.FileGroup` contains the collective behavior of `Moose.FAMIXFile`. The model-driven kernel of Moose uses this convention to navigate between domain classes (*i.e.* instance behavior) and helper classes (*i.e.* collective behavior). Moose's UI uses this convention to build dynamic menus for groups containing any kind of domain elements.

Using Moose's interactive UI, groups of objects are created by the user at runtime by selecting any kind of domain object. The user may, add objects the selection as he wants, and, may remove objects from the selection as he wants. Upon right-click, a dynamic menu is built for the selected objects, listing the collective behavior of the selection's least common supertype.

6.2 Refactoring of Moose

Based on above results, we carried out a refactoring of Moose's domain object hierarchy. Figure 3 illustrates Moose's domain object hierarchy, before and after the refactoring.

Before the refactoring, there are two parallel domain object hierarchies. The `MooseEntity` hierarchy, consisting of domain classes (*i.e.* instance behavior), and the `MooseGroup` hierarchy, consisting of helper classes (*i.e.* collective behavior). The helper classes on the left contained 20% of the domain logic, all subject to collective behavior. Helper classes and domain classes are link suing an arbitrary naming convention. `MooseGroup` duplicates most of `Collection`'s behavior. Any

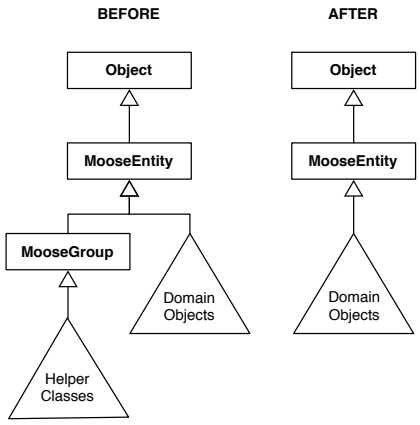

Fig. 3. Moose's domain object hierarchy, on the left, without collective behavior, on the right, using collective behavior. The helper classes on the left contained 20% of the domain logic, all subject to collective behavior. On the right, this logic is modeled as part of the domain object classes, using Collective method.

collection created and returned by the system or third-party requires special treatment, as it does not adhere to `MooseGroup`'s interface.

The refactoring consisted of following steps:

1. Copy all the helper classes's methods to the related domain objects's class, as *Collective method* .
2. Remove all code concerned with wrapping of general-purpose collections into helper classes or vice versa.
3. Manually fix any remaining reference to the helper classes.
4. Remove the helper classes, done.

After the refactoring, the system's domain logic is modeled in one class hierarchy. Collective behavior is used to model group-specific behavior of domain objects. For example, the code to show the System Complexity View of several classes, is implemented in `FAMIXClass group >> showSystemComplexity`.

7 Conclusion

Doug Lea wrote 1994 in [3]: "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". More than ten years later, groups are still not addressed first-class in object-oriented design. The concept of collective behavior, presented here, is meant to fills this gap. Collective behavior allow to associate custom behavior with collection instances, based on the type of its elements. The custom behavior is modeled as part of the

elements's class rather than the collections class. Due to this setup, collective behavior is orthogonal to the collection hierarchy.

The model of collective behavior has been presented, together with a proof-of-concept implementation and a case-study conducted to proof the claim, that collective behavior fills the gap between static and instance methods. The case-study revealed that 20% of the observed system's domain logic is subject to collective behavior, *i.e.* one out of five methods belongs to collective behavior.

Also, the case-study confirmed the importance of collective behavior's dynamic nature. In an interactive application, the content of user-selected groups is not known upfront. As elements join and leave the group, the least common supertype of its elements changes, and in the same turn, the associated custom behavior must also change dynamically at runtime.

Open issues, and thus subject to future work, are:

- Well-defined collective behavior on empty collections.
- Sound implementation using aspects, partial behavioral reflection, or any another well-performing and solid technique.
- Better tool support, *i.e.* IDE and version control extensions, to facilitate a better handling of Collective method at development.
- Combination of collective behavior with generics and/or array programming.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008). We would also like to express our thanks to Marcus Denker for the vivid discussions and feedback on behalf of collective behavior, and, to Phillippe Marschall and Klaus D. Witzel for spike-prototyping early implementations of collective behavior.

References

1. Craig Chambers. Predicate classes. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.
2. Stéphane Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.
3. Doug Lea. Objects in groups. submitted ECOOP '94SUNY at Oswego / NY Case Center, 1994.
4. Philippe Mougin 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.
5. David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 2007. To appear.