Group-Based Behavior Adaptation Mechanisms in Object-Oriented Systems

Patrick Rein, Stefan Ramson, Jens Lincke, Tim Felgentreff, and Robert Hirschfeld, Hasso Plattner Institute, University of Potsdam

// *Dynamic and distributed systems require behavior adaptations for groups of objects. Group-based behavior adaptation mechanisms scope adaptations to objects matching conditions beyond class membership. The specification of groups can be explicit or implicit.* //

AN IMPORTANT ASPECT of dynamic behavior adaptations is the static description of the scope of system parts to be adapted at runtime. Some *context-oriented programming* (COP) mechanisms limit behavior adaptations to classes.[1] However, systems with few classes, such as dynamic web applications, game engines, or service-based applications, require behavior adaptations based on more flexible scopes.

So, mechanisms have emerged that let programmers adapt the behavior of groups of objects (see Figure 1). These mechanisms originate from different use cases and programming languages. Here, we examine eight of them.

Adapting Objects beyond Classes

In the following example of a web-based presentation application, users can assemble new graphical objects by combining existing elements through drag and drop. During creation, contained objects can be dropped into and dragged out of their parent (see Figure 2).

One example adaptation is that users should be able to make a composition of objects persistent. A persistent composition behaves like a primitive object, preventing children from being dragged out. Users can also edit the composition by "opening it up" so that they can again drag the original child objects.

From a technical perspective, you can implement this behavior by maintaining two collections: `children` and `persistedChildren`. For this feature to be implemented, objects that are directly or indirectly in `persistedChildren` shouldn't exhibit the usual drag behavior. Expressed in code, this means that those objects just forward the `onDrag` behavior to their parents.

Although specifying this variation using class-based behavior adaptation mechanisms is possible, doing so isn't trivial. One way is to implement the new behavior in a common root class. However, this implementation assumes that all subclasses call the superclass implementation and don't add contradictory behavior, which doesn't always hold. Another solution is to

 0740-7459/17/$33.00 © 2017 IEEE

use metaprogramming to find and adapt each subclass. This solves the problem of overridden onDrag callbacks but introduces another problem: new objects of classes that are introduced after the adaptation will miss it.

How Group-Based Mechanisms Work

Group-based behavior adaptations explicitly describe behavior variations that are determined not by the object's class membership but by its group membership (see Figure 1). So, mechanisms for such adaptations comprise two parts: *group specification* and *behavior adaptation*.

There are two main ways to specify a group: programmers can explicitly add objects, or membership can be inferred from specific object properties. Mechanisms that employ explicit specification are easier to implement and have less impact on the language semantics. Mechanisms that employ implicit specification require less work from programmers to maintain groups.

By specifying a group, programmers provide a scope for the corresponding behavior adaptations: the adaptation is active while the object is a group member. Also, this group scope can be combined with other scoping mechanisms, such as dynamic scoping.

Programmers can describe a group's behavior adaptations in several ways that are also available in class-based adaptation mechanisms:

- COP layers,
- single methods to be added, or
- functions working on the whole group.

In general, behavior adaptations can both modify existing behavior and add new behavior.

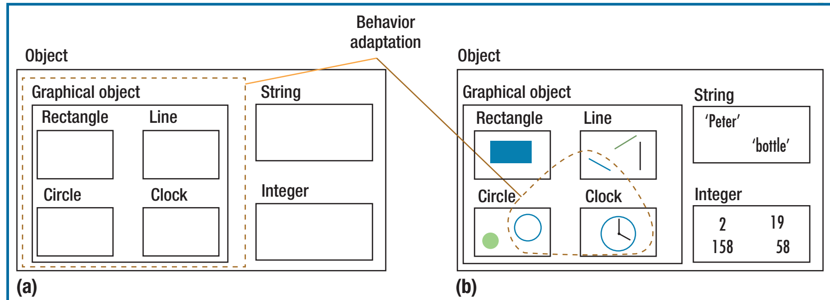


FIGURE 1. The possible scopes in (a) class-scoped behavior adaptation mechanisms and (b) group-based behavior adaptation mechanisms. Boxes denote classes, the inclusion of boxes denotes a subclass relation, and the dashed line denotes an example scope of an adaptation.

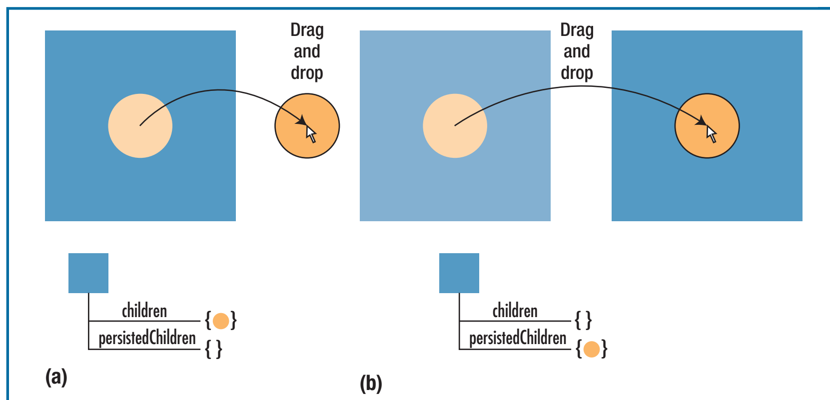


FIGURE 2. A behavior to be implemented (top) and the corresponding object graph (bottom). (a) Graphical objects in the children collection should handle the drag event themselves. (b) Graphical objects in any persistedChildren collection should forward the drag event to their parent element.

In our example application, we use *reactive object queries* (ROQs; for more on them, see the sidebar). We specify a group of objects declaratively through the following query in JavaScript:

```
let group = select(GraphicalElement,
    element => element.hasParent(
        (parent, child) => parent.
        persistedChildren.includes(child)));
```

The select function creates a new group that contains instances of the class GraphicalElement, which fulfills the condition expressed in the anonymous function passed as the second parameter. The method hasParent is a domain-specific call that checks whether an element has a parent that fulfills a certain criterion, again passed as an anonymous function.

The group is maintained automatically by the ROQ, which listens for object-state changes that could affect the query result. We then activate a behavior adaptation on the group through the following call,

GROUP-BASED BEHAVIOR ADAPTATION MECHANISMS

The following eight mechanisms illustrate the spectrum of implementations and use cases for group-based behavior adaptations. With these mechanisms, groups are specified either explicitly or implicitly.

EXPLICIT SPECIFICATION

Lively groups (LGs) are designed for interactive programming environments in which programmers create applications by combining objects.[1] LGs start off with plain objects and then add state and behavior to these objects individually. They let programmers describe behavior shared among the objects. The behavior is defined for a tag, which is assigned manually or programmatically to individual objects or groups of objects.

ContextErlang (CE) is an actor-based language that provides dynamic behavior adaptation.[2] Traditional object-oriented behavior adaptation mechanisms aren't a good fit for actor-based languages because there are no classes and message sends are asynchronous. CE lets programmers activate and compose behavior adaptations on individual actors through message sends. In that regard, it's similar to LGs, but it also supports synchronization of adaptations in an asynchronous environment.

Entity-component-system (ECS) is an architecture for scaling computer game development with regard to the myriad dynamic combinations of behaviors of game objects.[3] ECS splits a game object into three parts:

- The *entity* is the object's mere identity.
- *Components* constitute the data for each aspect of the object.
- The *system* constitutes the game logic working on multiple entities.

A group is defined by the set of components an entity must include. Systems are executed as part of the game loop and define the behavior common to all of that group's objects. Group maintenance is synchronized with the execution of systems through the game loop.

IMPLICIT SPECIFICATION

Predicated generic functions (PGFs) extend the dispatch mechanism for functional object-oriented environments, such as the Common Lisp Object System (CLOS).[4] A PGF is a family of methods, similar to CLOS generic functions. Each PGF implementation contains a set of predicates. When a PGF is invoked, the concrete implementation is selected on the basis of predicates that are checked against the parameters. In this way, it can be determined to which groups of objects the concrete implementation applies.

Active layers (ALs) are an implementation of context-oriented behavior adaptation that allows for user-defined scoping dimensions beyond dynamic scoping.[5] ALs achieve this by moving the layer composition into objects. During the method dispatch, `activeLayers` is called on the object. Programmers can override this method to change the layer composition. The behavior adaptations can be scoped to groups of objects but are prone to the subclassing issue we describe in the main article.

Reactive object queries (ROQs) aim primarily to free programmers from manually maintaining collections.[6] Through ROQs, programmers can declaratively describe collections of objects on the basis of their properties. The collections are maintained automatically. Because they denote relevant groups of objects, they can also be regarded as explicit scopes. To adapt behavior, ROQs allow for the activation of context-oriented programming (COP) layers for a collection's members.

Context groups (CGs) are a mechanism of the ServalCJ system that includes a generalization of activation models commonly found in COP systems.[7] Using CGs, programmers specify groups by describing combinations of activation conditions and activation scopes, which both describe the activation of COP layers. An object can then become a member of a CG and will thereby be subject to its behavior adaptations whenever a partial method is called.

Implied methods (IMs) support behavior reuse in scenarios the behavior's original developer didn't anticipate.[8] An IM consists of a set of conditions with which objects must comply and a method implementation that's added to the interface of matching objects. The conditions aim to reveal whether it's semantically correct to provide the method for an object.

which modifies the **onDrag** method's behavior for all its members:

```
group.layer({
  onDrag() {
    return this.parent.onDrag(event);
  }
});
```

This implementation directly expresses the desired functionality. Although it still explicitly mentions a class, it's used only as a representative of the set of all instances of the class and thus as the base set for the query. So, it works for all subclasses of **GraphicalElement**, even future ones. Additionally, whenever objects are added to or removed from **persistedChildren**, the adapted behavior will be activated or deactivated automatically.

Group-based behavior adaptation mechanisms are best applied when the environment doesn't provide classes (for example, actor- or prototype-based languages) and when various systems or users can contribute objects to the system (for example, distributed systems, service-oriented architectures, or complex interactive systems such as games). In these scenarios, a behavior adaptation's developer can't anticipate the objects' concrete classes but might still want variations to apply.

A Look at Explicit and Implicit Mechanisms

Here we examine how mechanisms can specify groups explicitly or implicitly by looking at the eight mechanisms we mentioned earlier. For more details on each mechanism, see the sidebar.

Explicit Group Specification

Lively groups (LGs), *ContextErlang* (CE), and *entity-component-system* (ECS) provide one designated group membership property per object. Programmers must manage this property. Besides assigning the property directly, programmers can use queries to select sets of objects to assign the property to. Either way, the corresponding code must be manually inserted at appropriate points in the control flow.

In our example application, programmers must manually add code to the system to explicitly set the group membership of graphical child objects whenever the parent object is persisted. Also, group membership must be updated if the parent is opened up again.

Although the implementations of explicit mechanisms have limited expressiveness, they require fewer features from the surrounding environment.

To enable the explicit assignment of group membership, the implementation must track each object's relevant property. This requires either modifying the root class or assigning the instance-specific state to objects, which occurs in LGs and CE. If neither is possible, the state might be tracked externally. Furthermore, when group membership changes, the behavior must be adapted. LGs and CE do this when the designated property changes.

For ECS, the group's behavior is added through an external function working on the group. So, the group must be updated before the next function is executed on the group.

Implicit Group Specification

These mechanisms automatically determine and manage an object's group membership on the basis of certain conditions. They're considered reactive because whenever the object state relevant to group membership changes, the group membership will have changed the next time the adapted behavior is called.

References

1. T. Felgentreff et al., "Lively Groups: Shared Behavior in a World of Objects without Classes or Prototypes," *Proc. Workshop Future Programming* (FPW 15), 2015, pp. 15–22.
2. G. Salvaneschi, C. Ghezzi, and M. Pradella, "ContextErlang: Introducing Context-Oriented Programming in the Actor Model," *Proc. 11th Ann. Int'l Conf. Aspect-Oriented Software Development* (AOSD 12), 2012, pp. 191–202.
3. S. Bilas, "A Data-Driven Game Object System," presentation at 2002 Game Developer Conf. (GDC 02), 2002; gamedevs.org/uploads/data-driven-game-object-system.pdf.
4. J. Vallejos et al., "Predicated Generic Functions: Enabling Context-Dependent Method Dispatch," *Proc. 9th Int'l Conf. Software Composition* (SC 10), 2010, pp. 66–81.
5. J. Lincke et al., "An Open Implementation for Context-Oriented Layer Composition in ContextJS," *Science of Computer Programming*, vol. 76, no. 12, 2011, pp. 1194–1209.
6. S. Lehmann et al., "Reactive Object Queries: Consistent Views in Object-Oriented Languages," *Companion Proc. 15th Int'l Conf. Modularity*, 2016, pp. 23–28.
7. T. Kamina, T. Aotani, and H. Masuharam, "Generalized Layer Activation Mechanism for Context-Oriented Programming," *Trans. Modularity and Composition*, LNCS 9800, Springer, 2016, pp. 123–166.
8. P. Rein et al., "Compatibility Layers for Interface Mediation at Run-time," *Companion Proc. 15th Int'l Conf. Modularity*, 2016, pp. 113–118.

ABOUT THE AUTHORS

PATRICK REIN is a doctoral researcher in the Software Architecture Group of the Hasso Plattner Institute, University of Potsdam. His research interests include live programming and the combination of personal information systems and programming systems. Rein received an MS in IT systems engineering from the Hasso Plattner Institute. Contact him at patrick.rein@hpi.uni-potsdam.de.

STEFAN RAMSON is a doctoral researcher in the Software Architecture Group of the Hasso Plattner Institute, University of Potsdam. His research interests include programming-language design and natural programming. Ramson received an MS in IT systems engineering from the Hasso Plattner Institute. Contact him at stefan.ramson@hpi.uni-potsdam.de.

JENS LINCKE is a member of the Software Architecture Group of the Hasso Plattner Institute, University of Potsdam. His research interests include live and explorative programming. Lincke received a PhD in IT systems engineering from the Hasso Plattner Institute. Contact him at jens.lincke@hpi.uni-potsdam.de.

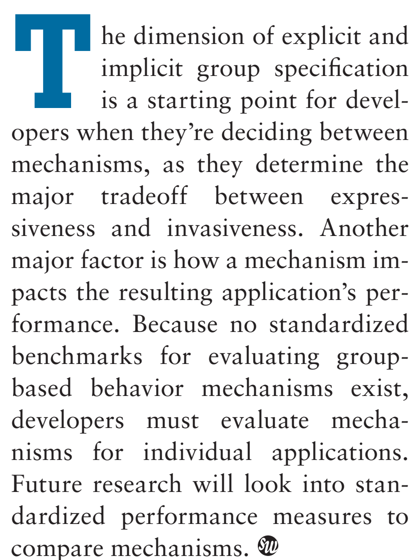
TIM FELGENTREFF is a senior software engineer at Oracle Labs. His research interests include programming-language extensions and high-performance dynamic-language virtual machines. He previously was a member of the Software Architecture Group of the Hasso Plattner Institute, University of Potsdam. Felgentreff received a PhD in IT systems engineering from the Hasso Plattner Institute. Contact him at tim.felgentreff@hpi.uni-potsdam.de.

ROBERT HIRSCHFELD leads the Software Architecture Group of the Hasso Plattner Institute, University of Potsdam. His research interests include dynamic programming languages, development tools, and runtime environments to make interactive programming more approachable. Hirschfeld received a PhD in computer science from the Ilmenau University of Technology. Contact him at robert.hirschfeld@hpi.uni-potsdam.de.

Predicated generic functions (PGFs) and *active layers* (ALs) let programmers directly define the conditions designating objects' group membership, thus emphasizing objects' local context. Other mechanisms such as ROQs, *context groups* (CGs), and *implied methods* (IMs) describe the conditions with regard to the complete object space.

Either way, implicit specification of group membership requires at a minimum that the surrounding environment supports introspection for individual objects. If a mechanism supports conditions on the complete object space, the environment also must support introspection on the complete space.

IMs, ALs, CGs, and PGFs influence the dispatch of the actual object behavior. Alternatively, the behavior can be changed eagerly (that is, directly at the moment) when the object becomes part of the group, as with ROQs. This eager change alters an object's interface, enabling reflection on the added behavior. Implementation of such an eager change requires tracking all changes to an object. When a change occurs, the conditions for group membership must be checked, and the object must be added or removed from the group.



The dimension of explicit and implicit group specification is a starting point for developers when they're deciding between mechanisms, as they determine the major tradeoff between expressiveness and invasiveness. Another major factor is how a mechanism impacts the resulting application's performance. Because no standardized benchmarks for evaluating group-based behavior mechanisms exist, developers must evaluate mechanisms for individual applications. Future research will look into standardized performance measures to compare mechanisms.

Reference

1. R. Hirschfeld, P. Costanza, and O. Nierstrasz, "Context-Oriented Programming," *J. Object Technology*, vol. 7, no. 3, 2008, pp. 125–151.

IEEE Software
TABLE OF CONTENTS
November/December 2017 Vol. 34 No. 6

FOCUS

CONTEXT-AWARE AND SMART HEALTHCARE

36 Guest Editors' Introduction
Recent Advances in Healthcare Software: Toward Context-Aware and Smart Solutions
Agusti Solanas, Jens H. Weber, Ayse Basar Bener, Frank van der Linden, and Rafael Capilla

42 Healthy Routes in the Smart City: A Context-Aware Mobile Recommender
Fran Casino, Constantinos Patsakis, Edgar Batista, Frederic Borràs, and Antoni Martínez-Ballesté

48 In the Pursuit of *Hygge* Software
Henrique Damasceno Vianna, Jorge Luis Victória Barbosa, and Fábio Pittoli

53 Crowd-Based Ambient Assisted Living to Monitor the Elderly's Health Outdoors
Ana Cristina Bicharra Garcia, Adriana Santarosa Vivacqua, Nayat Sánchez-Pi, Luis Martí, and José M. Molina

CONTEXTUAL-VARIABILITY MODELING

58 Guest Editors' Introduction
Modeling and Managing Context-Aware Systems' Variability
Kim Mens, Rafael Capilla, Herman Hartmann, and Thomas Kropf

64 Learning Contextual-Variability Models
Paul Temple, Mathieu Acher, Jean-Marc Jézéquel, and Olivier Barais

72 Dynamically Adaptable Software Is All about Modeling Contextual Variability and Avoiding Failures
Ismayle de Sousa Santos, Magno Luã de Jesus Souza, Michelle Larissa Luciano Carvalho, Thalisson Alves Oliveira, Eduardo Santana de Almeida, and Rossana Maria de Castro Andrade

78 Group-Based Behavior Adaptation Mechanisms in Object-Oriented Systems
Patrick Rein, Stefan Ramson, Jens Lincke, Tim Felgentreff, and Robert Hirschfeld

83 Context-Aware Software Variability through Adaptable Interpreters
Walter Cazzola and Albert Shaqiri