Instance Pointcuts:

An Aspect-Oriented Mechanism for Maintaining Object Categories

Kardelen Hatun Christoph Bockisch Mehmet Akşit
TRESE, University of Twente, 7500AE Enschede, The Netherlands
http://www.utwente.nl/ewi/trese/
{hatunk,c.m.bockisch,aksit}@ewi.utwente.nl

ABSTRACT
In the life-cycle of objects there are different phases. The phase in which an object currently is, affects how it is handled in an application; however phase shifts are typically implicit. In this study we propose an extension to the aspect-oriented language AspectJ with a new mechanism, called *instance pointcuts*, for categorizing objects according to events in their life-cycle; these events are selected with pointcut-like specifications. The selection criteria of instance pointcuts can be refined, e.g., to define a subset or super-set of an existing instance pointcut; and they can be composed, e.g., by boolean operations. We offer a means to access all objects currently selected by an instance pointcut from Java code, i.e., to be used in methods or advice bodies; and we expose the events of adding or removing an object from an instance pointcut by creating a join point that can be selected by regular pointcuts. Our approach improves modularity by providing a fine-grained mechanism and a declarative syntax to define and maintain object categories.

1. INTRODUCTION
In object-oriented programming (OOP), objects encapsulate state and behavior; objects also have a life-cycle, which means that the same object can play different roles at different times. Which role an object is currently playing can affect the object's own behavior or how it is handled. Typically the shift from one life-cycle phase to another is implicitly marked by events, e.g., passing an object from one client to another. Aspect-oriented programming (AOP) is a well-known technique for modularly implementing behavior applicable at events emitted from code that is not localized in a single type hierarchy. But current aspect-oriented languages do not offer declarative abstractions of object sets based on other criteria then the type. In this paper, we propose a new language mechanism for declaratively specifying life-cycle phases and for categorizing a set of objects which are currently in a specific phase. This declarativity allows us to give guarantees about these sets like subset relationships, as well as to perform compile-time checks like warning about sets that will always be empty.

As an example of different relevant phases in the life-cycle of objects, consider an online store application with "customer" objects representing the purchasers and "item" objects representing the products they add to their "shopping bags". We want to add discount policies for products that are treated by the customers in specific ways: For instance, a discount may apply for items that have been added to the shopping bag during the "happy hour". Thus, when calculating the price at check-out, we need to know which objects have been shopped within this hour. Categorizing objects according to criteria not directly supported by the programming language, such as the class they were initialized in, the method they are passed to as an argument, or (as in the example) the time at which they are passed to a method, requires invasively inserting bookkeeping code.

Aspect-oriented programming can be applied to separate this bookkeeping code from the business logic of the program. But in AOP, *pointcuts* select sets of so-called *join points* which are points in time during the execution of the program. Current aspect-oriented languages do not support a *declarative specification* of the objects belonging to a life-cycle phases; instead an *imperative implementation*, always following the same pattern, is required for collecting those objects. A consequence of such an imperative solution, besides all the negative effects of hand-writing boilerplate code, is that automatic reasoning becomes practically impossible.

To offer better support for processing objects according to their life-cycle phase, we propose to extend aspect-oriented programming languages. For this matter, we propose a new mechanism, called *instance pointcuts*, to select sets of objects based on the execution history. An instance pointcut definition consists of three parts: an identifier, a type which is the upper bound for all objects in the selected set, and a specification of relevant objects. The specification utilizes *pointcut expressions* to select events that define the begin and end of life-cycle phases and to expose the object. At these events, an object is added or removed from the set representing the instance pointcut.

New instance pointcuts can be derived from existing ones in several ways. Firstly, a new instance pointcut can be derived from another one by restricting the type of selected objects. Secondly, a *subset* or a *super-set* of an existing instance pointcut can be declared whereby the specification of the life-cycle phase is either narrowed down or broadened. Finally, instance pointcut declarations can be composed arbitrarily by means of boolean operators. In this paper we present instance pointcuts as an extension to AspectJ [1] and explain its semantics by showing a transformation from the new syntax to plain AspectJ.

2. MOTIVATION

Objects can be categorized by how they are used (passed as arguments to method calls, act as receiver or sender for method calls, etc.) and concerns of an application may be applicable only to objects used in a specific way. Therefore we must be able to identify and select those objects. We want to expose sets of objects belonging to the same category by means of a dedicated language construct such that the implementation of context-dependent concerns can explicitly refer to the category.

Below, we outline the architecture and design of an online store application. We use this scenario to give examples of categorizing objects according to how they are used and how to use these categories in the implementation of concerns. It must be noted that we intend to support the addition of *unanticipated* concerns, i.e., the program code is not prepared to support abstractions, like specific object categories, required by the new concerns. At the end of this section, we conclude requirements for solving the encountered challenges in these examples.

2.1 Example Architecture

An online shop is a sophisticated web application and objects of the same type can exist at different stages of the life-cycle. In Figure 1 the static structure of a simplified online shop is shown. When a new user logs in, `SessionManager` creates a `Customer` object to represent the user's session. A customer has a `ShoppingBag` and a `WishList`. The abstract class `Product` is a super-type of all products in the shop and represents product data. The `Product`'s subclasses are `BeautyProduct`, `BookProduct` etc. When the customer selects a product and clicks "add to shopping bag", a new `Item` instance is created and added to the `ShoppingBag` object; the `Item` instance contains the `Product` and quantity of that `Product`. A customer can add/remove items from his shopping bag. When the shopping is finished the `checkOut()` method is invoked on `Customer`, which returns an object of type `Order`. A `Customer` can also add or remove `Products` to/from his `WishList`; the `WishList` holds a list of `Products`, whereas `ShoppingBag` holds a list of `Items`.

2.2 Unanticipated Extensions

Let's assume a new requirement for applying a happy-hour discount is introduced. The discount should be applied at check-out to `Items` which have been added to a customer's shopping bag between certain hours. In order to realize this extension in an OO-approach, one needs to invasively change the system code: First we need to keep a set of items that

```
1  class ShoppingBag{
2    ...
3    Set<Item> happyItems = createSet();
4    public boolean addToShoppingBag(Product p, int amount)
5    {
6      Item item = ItemFactory.createItem(p,amount);
7      if('timing condition')
8        happyItems.put(item)
9      ...
10   }
11   public boolean removeFromShoppingBag(Item item, int amount)
12   {
13     ...
14     happyItems.remove(item);
15   }
16 }
17 class Customer{
18   ...
19   public checkOut()
20   {
21     ProductManager. ↙
            applyHappyHourDiscount(this.shoppingBag.happyItems);
22   }
23 }
```

Listing 1: A Java implementation of happy-hour discount rule

were added to a shopping bag within the timing condition. When the user finally checks out, we have to apply the discount to the items in the set. Listing 1 shows that we need to insert code in multiple places to satisfy the new requirement. First a set, called `happyItems`, is created (Line 1) in class `ShoppingBag`, to keep track of items that are added during the happy-hour. In the `addToShoppingBag` method the timing condition is checked, and when the condition is satisfied the created `Item` is added to `happyItems` (Lines 6–8). In the `Customer` class, more code is inserted to the `checkOut` method to apply the discount to the items in `happyItems` set, which is accessed through the `shoppingBag.happy` field (Line 21).

The OO-solution is scattered among `ShoppingBag` and `Customer` classes and tangled to multiple methods. Even for a single discount rule, the code for the discount concern and the book-keeping that comes with it creates cluttering. If we would like to apply multiple discount rules for the *check-out* event, this implementation style is clearly not suitable. An aspect-oriented implementation can offer a better solution by encapsulating the concern in an aspect. In Listing 2 the `newItem` pointcut (Lines 2–4) selects join-points where a customer adds a product to his bag: A new `Item` is created in the `addToShoppingBag` method. The advice is executed when `newItem` is matched; after returning the new `Item`, the timing condition is checked and if it holds it is added to `happyItems`.

We also define the `removeItem` pointcut (Lines 5–7) and advise it (Lines 15–18) to remove an item from `happyItems` if it is removed from a shopping bag. Finally, we define a pointcut for selecting the `Customer.checkOut()` join-point (Lines 8–9) and before the method is executed we apply the discount to items in `happyItems`.

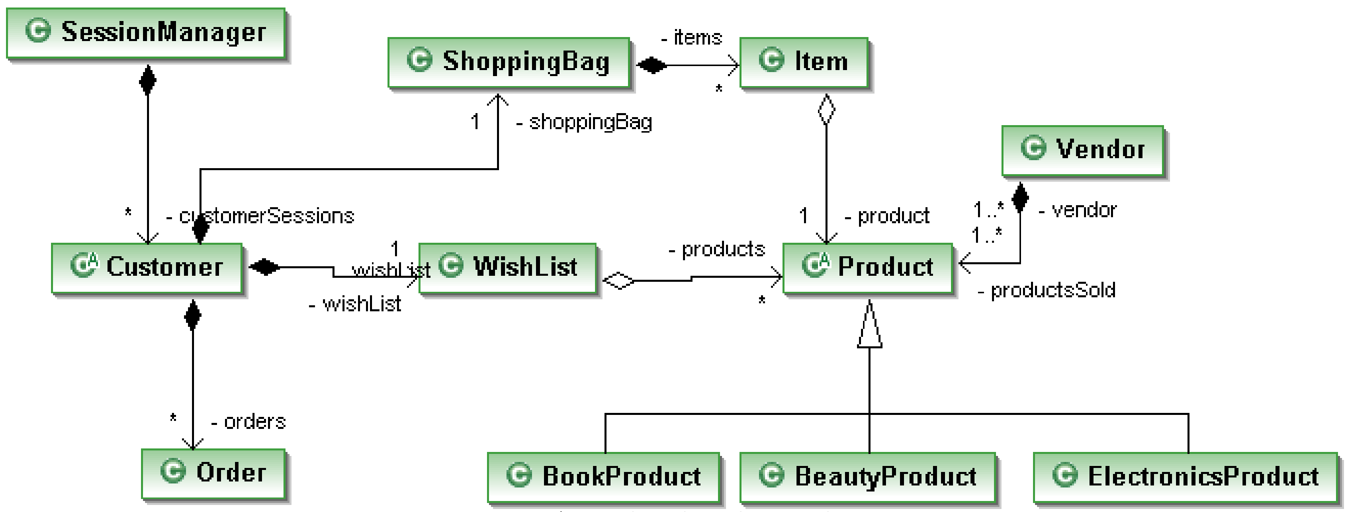


Figure 1: A simple online shop application

```
Set<Item> happyItems = createSet();
pointcut newItem(Item item):
      call(* ItemFactory.createItem(..)) && if('timing condition')
      && withincode(* ShoppingBag.addToShoppingBag(..));
pointcut removeItem(Item item):
   call(* ShoppingBag.removeFromShoppingBag(..)) &&
   args(item);
pointcut checkOut(Customer customer):
call(* Customer.checkOut()) && target(customer);

after() returning(Item item): newItem(item)
{
      happyItems.put(item);
}
after(Item item): removeItem(item)
{
   happyItems.remove(item);
}
before():checkOut(Customer customer)
{
   ProductManager.applyHappyHourDiscount(customer.items,
           happyItems);
}
```

Listing 2: An Aspectj implementation of happy-hour discount rule

Discussion and Requirements. AOP already helps to localize the concern, e.g., of a specific discount policy, and to add it without the need to modify existing code. But most of the implementation of the discount policy as shown in Listing 2 consists of boilerplate code. Only the definition of the pointcut and the advice to `checkOut` are concern-specific. Furthermore, reusing existing sets by specifying subsets or composing them is not conveniently supported at all; e.g., if we want to find the subset of `BeautyProducts` in a set of `Product` objects, we have to iterate over it and check instance types to create a new set.

In order to overcome the shortcomings of existing approaches, we need a way to declaratively select objects based their life-cycle phases, where the beginning and the end of a phase is marked by events. From the scenario described above and from our experience, we conclude the following requirements:

Requirement1:	A declarative way of reifying a set of objects by defining add/remove conditions must be provided.
Requirement2:	Since objects may recursively enter and exit a life-cycle phase, it must be counted how often the begin and end events have occurred for each object.
Requirement3:	Add/Remove expressions must select events and specify which object form the context of the event is selected.
Requirement4:	It must be possible to access the set of objects which currently comprises an object category and to be notified when the set changes.
Requirement5:	A declarative way to refine existing sets to subsets, supersets or compositions is required.
Requirement6:	Validity should be ensured for subset and superset relationships, and composition in general, either by construction or through compile-time checks.

3. INSTANCE POINTCUTS

Instance pointcut is a declarative language construct that is used to reify and maintain a set of objects of a specified type, with the ability to select them over a period marked by events in their life-cycle, modularizing the object selection concern and making it declarative.

In the remainder of this section, we will explain instance pointcuts in detail. We have implemented a prototype by extending AspectJ. Throughout the section code examples of instance pointcuts will be given in extended AspectJ.

3.1 Features

A concrete instance pointcut definition consists of a left hand-side and a right-hand side. On the left-hand side the pointcut's name and a type reference of interest is declared. An instance pointcut does not declare pointcut parameters; it has a single implicit parameter called `instance` of the declared type. On the right-hand side a pointcut expression selects the desired events from join points and then binds the exposed object (represented by `instance` parameter) as a member of the instance pointcut's set. An instance pointcut is a *static aspect member* (Listing 3).

```
aspect MyAspect{
  static instance pointcut foo<Object> ...
}
```

Listing 3: An instance pointcut declaration in an aspect

3.1.1 Add/Remove Expressions

Instance pointcuts implicitly perform add to set and remove from set operations, when certain events are matched. An instance pointcut expression, which consists of two sub-expressions separated by the `UNTIL` keyword, defines the events and matching conditions. One of the sub-expressions is the *add expression* that selects events which expose the instances to be added to the instance pointcut set; an event selected by the add expression marks the beginning of the life-cycle phase of interest. After the `UNTIL` clause an optional sub-expression called the *remove expression* can be defined, which specifies when to remove which object: it selects the event that marks the end of the life-cycle phase of interest.

In AspectJ join points mark *sites* of execution; a join point by itself does not define an event. Pointcut expressions select join points and pointcuts are used with advice specifications to select a particular event in that join point. The shortcomings of this region-in-time model is discussed by Masuhara et al. in [2]. We combine pointcut expressions with advice specifiers and obtain expression elements. Each expression element contains a pointcut expression, which matches a set of join points. Then, from these join points, according to the advice specifier the before or after events are selected. Both add and remove expressions contains at least one *expression element* and at most two. The expression elements can be one of two types, *before element* and an *after element.*

The grammar definition in Figure 2 shows part of the instance pointcut syntax. The $\langle pointcut-expression\rangle$ rule

$\langle$*instance pointcut*$\rangle$ ::= ‘`instance pointcut`’ $\langle$*name*$\rangle$ ‘<’
    $\langle$*instance-type*$\rangle$ ‘>’ ‘`:`’ $\langle$*ip-expr*$\rangle$ (‘`UNTIL`’ $\langle$*ip-expr*$\rangle$)?

$\langle$*ip-expr*$\rangle$ ::= $\langle$*after-event*$\rangle$ ‘`||`’ $\langle$*before-event*$\rangle$
  | $\langle$*before-event*$\rangle$ ‘`||`’ $\langle$*after-event*$\rangle$
  | $\langle$*after-event*$\rangle$
  | $\langle$*before-event*$\rangle$

$\langle$*after-event*$\rangle$ ::= ‘`after`’ ‘`(`’$\langle$*pointcut-expression*$\rangle$‘`)`’

$\langle$*before-event*$\rangle$ ::= ‘`before`’ ‘`(`’$\langle$*pointcut-expression*$\rangle$‘`)`’

Figure 2: Grammar definition for instance pointcuts

represents an AspectJ pointcut expression with the restriction that every pointcut expression there must be a binding predicate (args, target etc.) that bind the `instance` parameter and an extension that allows the use of `returning` clause as a binding predicate. The `returning` clause binds the returned variable by a method or a constructor and in AspectJ it can only be used in advice specifications. In an instance pointcut expression, it is only possible to `OR` a before event with an after event. The *before* clause selects the start of executing an operation (i.e., the start of a join point in AspectJ terminology) and the *after* clause selects the end of such an execution. For two operations that are executed sequentially, the end of the first and the start of the second operation are treated as two different events. Thus, the before and after clauses select from two disjoint groups of events and the conjunction of a before and an after clause will always be empty.

```
static instance pointcut customers<Customer>:
  after(call(SessionManager.createCustomer(..)) &&
        returning(instance))
  UNTIL
  before(call(SessionManager.destroyCustomer(..)) &&
        args(instance))
```

Listing 4: A basic instance pointcut declaration with add and remove expressions

The instance pointcut in Listing 4 shows a basic example. The left-hand side of the instance pointcut indicates that the pointcut is called `customers` and it is interested in selecting `Customer` objects. On the right hand side, there are two expressions separated by the `UNTIL` keyword. The first one is the add expression. It selects the return event of the method `createCustomer` and from the context of this event it exposes the returned `Customer` object with the `returning` clause and binds it to the `instance` parameter. If the `instance` parameter is bound by the `returning` clause then it has to be in an after event. The second one is the remove expression and it selects the before event `destroyCustomer` call and exposes the `Customer` instance in the method arguments and binds it to the `instance` parameter with `args` pointcut.

Note that instance pointcuts do not keep objects alive, as instance pointcuts are non-invasive constructs, which do not affect the program execution in any way. So even if the remove expression was not defined for the `customers` instance pointcut, when the `Customer` instances are collected by the garbage collector, they are removed from the set.

3.1.2 Multisets

An instance pointcut reifies an object set as a *multiset.* Multisets allow multiple appearances of an object. The instance pointcut shown in Listing 5 selects `Product` instances, which are added to the wishlist of a `Customer`. The remove expression removes a `Product` instance if it is removed from the wish-list of a `Customer`. With this pointcut we would like to represent the currently wishlisted products. Multiset makes sure that `Products` can be added for each add to wishlist operation. When the same product is added to wishlists of two different customers, and if one of them removes it from the wishlist, then there's still one entry of that instance is left in the set. If instance pointcuts only supported a set then as soon as a customer removes a product from his wishlist, its only copy will be removed and it will appear removed from all customers' wishlists.

```
static instance pointcut multi_wish<Product>:
  before(call(Customer.addToWishList(..)) &&
          args(instance))
  UNTIL
  before(call(Customer.removeFromWishList(..)) &&
          args(instance))
```

Listing 5: An instance pointcut utilizing multiset property

3.2 Refinement and Composition

Instance pointcuts can be referenced by other instance pointcuts, they can be refined to create a subset or broadened to create a superset.

```
static instance pointcut lowSelling<Product>:
  after(call(* OnlineShop.setLowSelling(..)) &&
          args(instance));
```

Listing 6: An instance pointcut selecting low-selling products

3.2.1 Referencing and Type Refinement

Instance pointcuts are referenced by their name. Optionally the reference can also take an additional statement for *type refinement*, which selects a subset of the instance pointcut of refined type. Type refinements require that the refined type is a subtype of the original instance type. For example the instance pointcut `lowSelling` (Listing 6) can be refined as in the following:

```
static instance pointcut lowSellingBeauty<BeautyProduct>:
  lowSelling<BeautyProduct>
```

This selects the subset of `BeautyProduct` instances from the set of `Product` instances selected by `lowSelling`. Note that this notation will also select subtypes of `BeautyProduct`.

3.2.2 Instance Pointcut Composition

It is also possible to compose instance pointcuts to obtain a new instance pointcut based on the component instance pointcuts. In Figure 3, an extended version of the grammar definition is shown. A new instance pointcut can be defined by declaring a reference expression. The reference expression can be a single instance pointcut reference or it can be defined as the composition of other instance pointcuts. Furthermore, expression elements can be composed with the reference expression. In order to obtain a more refined set && operation is used and for a broader set ‖ is used.

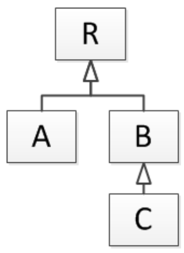
```
⟨instance-pointcut⟩ ::= 'instance pointcut'
    ⟨name⟩ '<' ⟨instance-type⟩ '>' ':' ((⟨ip-ref-expr⟩
    ⟨boolean-operator⟩)? (⟨ip-expr⟩ ('UNTIL' ⟨ip-expr⟩)?)?

⟨boolean-operator⟩ ::= '&&'
  | '||'

⟨ip-ref-expr⟩ ::= ⟨ip-ref⟩
  | ⟨ip-ref-expr⟩ ⟨boolean-operator⟩ ⟨ip-ref-expr⟩

⟨ip-ref⟩ ::= ⟨name⟩
  | ⟨name⟩('<' ⟨refined-instance-type⟩ '>')?
```

Figure 3: Syntax for instance pointcut composition

The type of an instance pointcut must correspond to the type of the composed instance pointcuts. For a composition of two instance pointcuts, the type of the composed one can be determined depending on the relation of the types of the component instance pointcuts and the type of composition. Consider the representative type hierarchy in Figure 4a: `R` is the root of the hierarchy with the direct children `A` and `B` (i.e., these types are siblings); `C` is a child of `B`. Table 4b shows four distinct cases: Either the type of one of the instance pointcuts is a super type of the other one's type (second row), or both types are unrelated (third row); and the composition can either be && (third column) or ‖ (fourth column).

When composing two instance pointcuts with types from the same hierarchy, the type of the composition is the more specific type (`C` in the example) for an && composition and the more general type (`B`) for an ‖ composition. When composing two instance pointcuts with sibling types, for the && operation the resulting composition cannot select any types since the types `A` and `B` cannot have a common instance. The ‖ operation will again select a mix of instances of type `A` and `B`, thus composed instance pointcut must have the common super type, `R` in the example.

Semantics. The composition of two instance pointcuts is computed by composing their instance pointcut expressions. An instance pointcut can have at most four expression elements; before add (*ba*) and after add (*aa*) form the add expression and, before remove (*br*) and after remove(ar) form the remove expression. Let us define an instance pointcut as $IP = ba_{IP} \parallel aa_{IP}$ UNTIL $br_{IP} \parallel ar_{IP}$. Each element contains a pointcut expression referred with the notation $ba_{IP}\_expression$.



(a) A simple type hierarchy

pc1 type	pc2 type	&&	∥
B	C	C	B
B	A	∅	R

(b) Instance pointcut compositions and effect on the captured instance type

Figure 4: An example to illustrate composition's effect on types

$$IP1 = ba_{IP1} \parallel aa_{IP1} \tag{1}$$
$$IP2 = ba_{IP2} \parallel aa_{IP2} \tag{2}$$

Let us define a new instance pointcut as; $IP3 = IP1 \;\&\&\; IP2$. By distributivity this operation results in the following formulation:

$$\begin{aligned} IP3 \rightarrow \text{add expression} = {} & (ba_{IP1} \;\&\&\; ba_{IP2}) \\ & \parallel (ba_{IP1} \;\&\&\; aa_{IP2}) \\ & \parallel (aa_{IP1} \;\&\&\; aa_{IP2}) \\ & \parallel (aa_{IP1} \;\&\&\; ba_{IP2}) \end{aligned} \tag{3}$$

The composition of before and after event via && will be empty as explained at the beginning of this section. Composing two events of the same type with a logical operator is formulated as follows, where event is either *before* or *after*:

$$event(exp_1) \;\&\&\; event(exp_2) = event(exp_1 \;\&\&\; exp_2)$$
$$event(exp_1) \parallel event(exp_2) = event(exp_1 \parallel exp_2)$$

Note that we do not analyze expressions to verify their composition and leave that to the AspectJ compiler. For example AspectJ does not allow use of different binding predicates in the composed expression and this is also a limitation of instance pointcut composition.

Finally IP3 becomes:

$$IP3 = \overbrace{(ba_{IP1} \;\&\&\; ba_{IP2})}^{ba_{IP3}} \parallel \overbrace{(aa_{IP1} \;\&\&\; aa_{IP2})}^{aa_{IP3}} \tag{4}$$

If IP1 and IP2 define remove expressions, then the composed instance pointcut IP3 will become:

$$\begin{aligned} IP3 = {} & \overbrace{(ba_{IP1} \;\&\&\; ba_{IP2})}^{ba_{IP3}} \parallel \overbrace{(aa_{IP1} \;\&\&\; aa_{IP2})}^{aa_{IP3}} \\ & \text{UNTIL} \\ & \underbrace{(br_{IP1} \parallel br_{IP2})}_{br_{IP3}} \parallel \underbrace{(ar_{IP1} \parallel ar_{IP2})}_{ar_{IP3}} \end{aligned} \tag{5}$$

Notice that the remove expressions are applied ∥ instead of &&. If we narrow down the removal criteria by && then it is possible that some objects which are added will not be removed. By keeping the removal criteria for both of the composed instance pointcuts, we ensure that all the added objects can be removed. So we can say that the && and the ∥ operations only effect the expression elements in the add expression.

If we define another instance pointcut as $IP4 = IP1 \parallel IP2$, after performing the above formulations we get:

$$\begin{aligned} IP4 = {} & \overbrace{(ba_{IP1} \parallel ba_{IP2})}^{ba_{IP4}} \parallel \overbrace{(aa_{IP1} \parallel aa_{IP2})}^{aa_{IP4}} \\ & \text{UNTIL} \\ & \underbrace{(br_{IP1} \parallel br_{IP2})}_{br_{IP4}} \parallel \underbrace{(ar_{IP1} \parallel ar_{IP2})}_{ar_{IP4}} \end{aligned} \tag{6}$$

Using the same composition principles, instance pointcuts can also be composed with expression elements, to refine or broaden the instance pointcut scope. For the instance pointcut IP5 which is defined as:

$$\begin{aligned} IP5 = {} & IP1 \,||\, (ba_{new}\; UNTIL\; ar_{new}) \\ = {} & \overbrace{(ba_{IP1} \parallel ba_{new})}^{ba_{IP5}} \parallel \overbrace{(aa_{IP1})}^{aa_{IP5}} \\ & \text{UNTIL} \\ & \underbrace{(br_{IP1})}_{br_{IP5}} \parallel \underbrace{(ar_{IP1} \parallel ar_{new})}_{ar_{IP5}} \end{aligned} \tag{7}$$

Let us illustrate these concepts with a concrete example. The first instance pointcut (`lowSelling`) shown in Listing 7 selects the `Products` which are marked as *low-selling* by the `OnlineShop` class and remove those products which are discontinued by the `Vendor`. The second instance pointcut (`wishList`) selects the `BookProducts` which are added to the wishlist of a `Customer` and removes them when they are out of the wishlist.

The new instance pointcut `composition` is the result of an ∥ operation between `lowSelling` and `wishList`. Then the add and remove expressions of `composition` is created by factoring out the `args` pointcut and the composed instance pointcut expression is shown in Listing 8.

Similarly we can define a new instance pointcut (Listing 9),

```
static instance pointcut lowSelling<Product>:
  after(call(* OnlineShop.setLowSelling(..)) &&
        args(instance))
  UNTIL
  before(call(* Vendor.discontinue(..)) &&
        args(instance));

static instance pointcut wishList<BookProduct>:
  after(call(* Customer.addToWishList(..)) &&
        args(instance))
  UNTIL
  before(call(* Customer.removeFromWishList(..)) &&
        args(instance)) ;

static instance pointcut composition<Product>:
  lowSelling || purchase;
```

Listing 7: Two instance pointcuts composed to obtain a new one

```
static instance pointcut composition<Product>:
  after((call(* OnlineShop.setLowSelling(..)) || call(* ↙
        Customer.addToWishList(..))) &&
         args(instance))
  UNTIL
  before((call(* Vendor.discontinue(..)) || call(* ↙
        Customer.removeFromWishList(..))) &&
         args(instance))
```

Listing 8: The new instance pointcut after composition

this time by composing an instance pointcut with an expression element, then the `composition2` instance pointcut has the pointcut expression shown in Listing 10.

```
static instance pointcut composition2<Product>:
  lowSelling &&
  (after(if(instance.getPrice() > 10))
  UNTIL
  before(call(* OnlineShop.removeFromShop(..)) &&
        args(instance));
```

Listing 9: An instance pointcut and an expression is composed

```
static instance pointcut composition2<Product>:
  after(call(* OnlineShop.setLowSelling(..)) &&
        args(instance) &&
        if(instance.getPrice() > 10))
  UNTIL
  before((call(* Vendor.discontinue(..)) ||
        call(* OnlineShop.removeFromShop(..))) &&
        args(instance));
```

Listing 10: The composition2 instance pointcut after composition

3.2.3 Subset and Superset Relationships

There are two mechanisms to define relationships between instance pointcuts, *subsetting* and *supersetting*. The declaration of these relationships impose some restrictions on the *referencing* instance pointcut. In Figure 5 the relationship syntax is shown.

⟨*instance-pointcut*⟩ ::= ...
  | 'instance pointcut' ⟨*name*⟩ ('<' ⟨*instance-type*⟩ '>')? ⟨*relationship*⟩ ':' ⟨*ip-expr*⟩ ('UNTIL' ⟨*ip-expr*⟩)?
  | 'instance pointcut' ⟨*name*⟩ ('<' ⟨*instance-type*⟩ '>')? ⟨*relationship*⟩ ':' ⟨*ip-ref*⟩

⟨*relationship*⟩ ::= 'subsetof' ⟨*ip-ref*⟩
  | 'supersetof' ⟨*ip-ref*⟩

Figure 5: Grammar definition for relationship declarations

For the declaration **instance pointcut** subsetIP1 **subsetof** IP1:..., the subset relationship imposes that the instance set created by `subsetIP1` has to be the subset of the instance set of `IP1`. When the *instance type* is not declared like in the example then it is inferred from the referenced instance pointcut, `IP1`. It is also possible to declare subsets by using type refinement. For example **instance pointcut** ↙ subsetIP1<SubType> **subsetof** IP1: IP1<SubType> creates a subset of `IP1` by selecting the instances of type `SubType`. For the subset relationship the referencing type must declare a co-variant type of the referenced instance pointcut. For the superset declaration **instance pointcut** supersetIP1 **supersetof** ↙ IP1:..., the superset relationship imposes that the instance set created by `supersetIP1` contains the set of `IP1`. This rule holds even if the scope of `supersetIP1` is narrowed. When the instance type is not declared then the instance type of `IP1` is copied. For the superset relationship referencing instance pointcut must declare a contra-variant type.

The right-hand side of a relationship declaration can be a full instance pointcut expression, formed by add and remove expressions. For subset relationships, && composition rule and for superset relationships, ∥ composition rule is applied, composing this expression with the referenced instance pointcut's expression.

Subset and superset relationships enforce consistency between instance pointcuts and reduce redundancy. Being able to select subsets/supersets of instance pointcuts eliminates the need for defining separate instance pointcuts, which may result in erroneous selections. Since the subsets only support subtypes and supersets support only the supertypes of the referenced instance pointcut's type, it works with the system's type hierarchy in a natural manner.

3.2.4 Checking

Instance pointcuts also have checking mechanisms to warn the user for potential errors and enforcing some constraints. In some cases it can be possible to identify if a set is empty or not during compile time, for example when the pointcut expressions defined in expression elements of the instance pointcut do not match any join points. For the subset and superset relationships, type checking is performed to see if they satisfy the typing restrictions. Currently more advanced checks and verifications are still future work.

3.3 Using Instance Pointcuts

Up to now we have explained the syntax and semantics for instance pointcuts. In this section we will explain how to use instance pointcut in the context of an AO languge, namely, AspectJ. The instance pointcut defined in Listing 11 maintains a set of `Products` that are currently out of stock.

```
aspect MyAspect{
  static instance pointcut outOfStock<Product>:
    after(call(∗ Product.outOfStock(..)) && target(instance)) ↙
           UNTIL after(call(∗ Vendor.stock(..)) && ↙
           args(instance));
  ...
}
```

Listing 11: An instance pointcut for out of stock products

3.3.1 Set Access

Instance pointcuts reify a set and this set can be accessed through the identifier. When an instance pointcut is referenced in Java code, it means that its set is accessed. Then the *read* methods of the multiset interface can be used to retrieve objects from the set. Write methods are not allowed since it creates data inconsistencies and it may result in concurrent modification exceptions.

```
1 public static double calculateDamages()
2 {
3   double damage = 0;
4   for(Product p: MyAspect.outOfStock)
5     damage = damage + p.getPrice();
6   return damage;
7 }
```

Listing 12: Calculate a damage estimate for out of stock products

In Listing 12, a method called `calculateDamages` is defined in `MyAspect`. This method calculates a rough estimate of the shop's damages when products are out of stock. On line 4, the for loop iterates over the `outOfStock`'s set, which is accessed as a static field.

3.3.2 Set Monitoring

An instance pointcut definition defines two set change events, an add event and a remove event. In order to select the join points of these events, every instance pointcut definition automatically has two implicit regular pointcuts. These implicit pointcuts have the following naming conventions, $\langle name \rangle$*_add*, $\langle name \rangle$*_remove*, where $\langle name \rangle$ is the name of the instance pointcut. These pointcuts allow the user to access the set change event and the object to be added or removed. In Listing 13, a before advice using the `outOfStock_add` pointcut is shown. When a product is marked out of stock then it is added to the set, this advice uses the `Product` instance to be added and notifies the related `Vendor` that the product is out of stock.

```
before(Product p): outOfStock_add(p)
{
  OnlineShop.notifyVendor(p.getVendor, STOCK_MSG);
}
```

Listing 13: Set monitoring pointcut used to notify vendors

3.4 Compilation of Instance Pointcuts

We have implemented the instance pointcut language with the EMFText language workbench. For this purpose, we have defined the AspectJ grammar based on the one published in **???** and extended it with the grammar for instance pointcuts which was presented interspersed with the rest of this section. During compilation, the instance pointcut definitions are transformed into semantically equivalent source code, the remaining definitions are preserved as they are; the result is compiled with the AspectJ compiler. A goal for our compiler implementation is to support modular compilation. This means to compile an aspect with instance pointcuts that depend on instance pointcuts defined in other aspects, it must be sufficient to know their declaration (i.e., the name and type); it should not be necessary for the compiler to know the actual expression or the referenced instance pointcuts.

The Listing 14 shows the code which is generated by our compiler for managing the data of an instance pointcut; the variables *${Type}* and *${ipc}* stand for the instance pointcut's type and name. First, a `WeakHashMap` is defined for storing the instances currently selected by an instance pointcut (cf. line 1). We use weak references to avoid keeping objects alive which are not reachable from the base application anymore. The keys of the map are the objects which are selected by the instance pointcut; the mapped value is a counter of how often the object has been selected by the instance pointcut. The generated method *${ipc}* returns all objects which are currently mapped (cf. lines 3–5).

Furthermore, methods are generated to increase or decrease the counter of selected objects; if an object does not have an associated counter yet or the counter reached zero, the object is added to or removed from the map, respectively (cf. lines 7–16). After having performed their operation, both method invoke an empty method, passing the added or removed object. We generate a public, named pointcut selecting these calls, exposing the respective events (cf. lines 22 and 23).

```
1  private static WeakHashMap<${Type}, Integer> ${ipc}_data ↙
         = new WeakHashMap<${Type}, Integer>();

3  public static Set<${Type}> ${ipc}() {
4    return Collections.unmodifiableSet(${ipc}_data.keySet());
5  }

7  public static void ${ipc}_addInstance(${Type} instance) {
8    increase counter associated with instance by the ↙
           ${ipc}_data map
9    ${ipc}_instanceAdded(instance);
10 }

12 public static void ${ipc}_removeInstance(${Type} instance) {
13   decrease counter associated with instance by the ↙
           ${ipc}_data map
14   if the counter reaches 0, remove instance from the map
15   ${ipc}_instanceRemoved(instance);
16 }

18 private static void ${ipc}_instanceAdded(${Type} instance) {}

20 private static void ${ipc}_instanceRemoved(${Type} instance) {}

22 public pointcut ${ipc}_add(${Type} instance) : call(private ↙
         static void Aspect.${ipc}_instanceAdded(${Type})) && ↙
```

```
        args(instance);
public pointcut ${ipc}_remove(${Type} instance) : call(private ↙
        static void Aspect.${ipc}_instanceRemoved(${Type})) ↙
        && args(instance);
```

Listing 14: Template of generated code for instance set management.

Next, these bookkeeping methods have to be executed at the events specified in the instance pointcut expression. At first sight, realizing this by generating AspectJ pointcuts and advice seems to be a solution. However, the way AspectJ handles binding values and restricting their types in pointcuts prevents a modular compilation of instance pointcuts:

An instance pointcut expression can be transformed into several named AspectJ pointcuts which bind a single value, the *instance*. However, in AspectJ the result value of a join point is bound by the **after returning** keyword in the advice definition instead of in the pointcut definition. Therefore, the generated AspectJ code depends on which value is bound; this means that it must be known which value is bound when making a reference to a pointcut. It is also not possible to work around this, using AspectJ's reflective **thisJoinPoint** keyword, as this does not expose the result value at all. Another, similar limitation is that AspectJ does not allow to narrow down the type restriction for the bound value of a referred pointcut. Thus, in order to be able to transform an instance pointcuts definition to AspectJ, it is necessary to know the definitions of all instance pointcuts it refers to and to inline their definition.

For these reasons , our current implementation is based on the ALIA4J[1] [?, ?] implementation approach and framework for advanced dispatching languages. At its core, ALIA4J contains a meta-model of advanced dispatching declarations, called , and a framework for execution environments that handle these declarations, called *FIAL*. The term advanced dispatching refers to late-binding mechanisms including, e.g., predicate dispatching and pointcut-advice mechanisms.

An instance pointcut, generally consists of four underlying pointcut definitions: specifying join points (1) *before* or (2) *after* which an instance is to be *added* to the selected instances; and specifying join points (3) *before* or (4) *after* which an instance is to be *removed*. For each pointcut definition, we generate a method that creates a corresponding LIAM model; the methods are called *${ipc}*_add_before, *${ipc}*_add_after, *${ipc}*_remove_before, and *${ipc}*_remove_after.

In LIAM, a *Specialization* can represent a partial AspectJ pointcut and a full pointcut expression can be represented as the disjunction of a set of *Specializations* (discussed in detail elsewhere [?]). Figure 6 shows the meta-model for a specialization in ALIA4J consisting of three parts. A *Pattern* specifies syntactic and lexical properties of matched dispatch site (or join point shadows). The *Predicate* and *Atomic Predicate* entities model conditions on the dynamic state a dispatch depends on (dynamic pointcut designators). The *Context* entities model access to values like the called

[1]The Advanced-dispatching Language Implementation Architecture for Java. See `http://www.alia4j.org/alia4j/`.

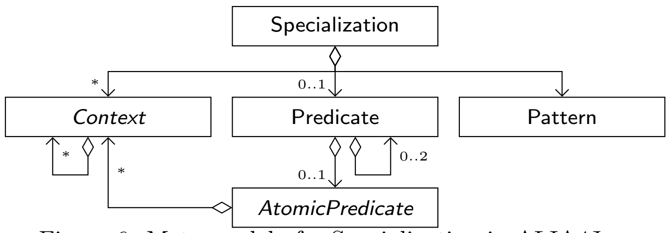


Figure 6: Meta-model of a Specialization in ALIA4J.

object or argument values. Contexts which are directly referred to by the Specialization are exposed to associated advice.

Depending on the definition of the instance pointcut, LIAM models of the underlying pointcuts have to be created in different ways. For pointcut expressions that are directly provided, we use a library function provided by ALIA4J which takes a String containing an AspectJ pointcut as input. We have extended this library to also understand the **returning** pointcut designator. All these parts are optional; a missing part can be represented as an empty set of Specializations in LIAM.

```
static instance pointcut ${ipc}<${Type}>:
ipc1<${Type2}>
before(${pc_add_before}) after(${pc_add_after}) UNTIL
before(${pc_remove_before}) after(${pc_remove_after};
```

Listing 15: Example of an instance pointcut using composition and type refinement

For the example instance pointcut presented in listing 15, we show the code generated for the method creating the LIAM model for the add/before pointcut in listing 16; the other methods are generated analogously. Line 4 shows the transformation of an AspectJ pointcut into a set of Specializations in the LIAM meta-model by passing the pointcut as a String—represented by *${pc_add_before}* in 15 and 16—the above mentioned library function.

What is generated from an instance pointcut reference, is shown in line 6: the LIAM model of the corresponding underlying pointcut of the referenced instance pointcut is invoked (cf. line6). If there is a type refinement defined for this instance pointcut reference, we add a restriction to the retrieved Specializations. The implementation of the addTypeConstraint method returns a copy of the passed Specializations where the Predicate is extended with a test that the bound value is an instance of *${Type2}* (cf. line 7). Finally, the full LIAM model of the pointcut definition is retrieved by forming the disjunction of the LIAM model for the direct pointcut expression and the LIAM model for the instance-pointcut-reference expression (cf. line 9)—which can itself consist of conjunctions and disjunctions of multiple instance-pointcut references.

```
1  public static Set<Specialization> ${ipc}_add_before() {
2    private static Set<Specialization> ${ipc}_add_before;
3    if (${ipc}_add_before == null) {
4      Set<Specialization> simpleIPExpr = ↙
           Util.toSpecializatons("${pc_add_before}", ${Type});

6      Set<Specialization> ipRefExpr = ipc1_add_before();
```

```
 7        ipRefExpr = Util.addTypeConstraint(ipRefExpr, ${Type2});

 9        ${ipc}_add_before = Util.orSpecializations(ipRefExpr, ↙
                simpleIPExpr);
10     }
11     return ${ipc}_add_before;
12   }
```
Listing 16: Generated code for creating the LIAM model for the add/before pointcut of the example instance pointcut.

The disjunction performed in line 9 of listing 16 corresponds to the usage of the operator in line 2, listing 15. When the && operator is used instead, the method andSpecialization is used in the generated code, but only in creation of the pointcuts for adding instances to the map. For the removal pointcuts, the simple pointcut expression and the instance pointcut reference expression is always *or*ed.

Finally, the created LIAM models of the pointcuts must be associated with advice invoking the add or remove method for the instance pointcut. In a LIAM model this is achieved by defining an Attachment, which roughly corresponds to a pointcut-advice pair. An attachment refers to a set of Specializations, to an *Action*, which specifies the advice functionality, and a *Schedule Information*, which models the time relative to a join point when the action should be executed, i.e., "before," "after," or "around".

Listing 17 shows the generated code for creating and deploying the bookkeeping Attachments. The first Attachment uses the set of Specializations returned by the ${ipc}add_before method (cf. line 4) and specifies the ${ipc}_addInstance method as action to execute at the selected join points (cf. line 5). As relative execution time, the Attachment uses a "SystemScheduleInfo"; this is provided by ALIA4J for Attachments performing maintenance whose action should be performed before or after all user actions at a join point, such that all user actions observe the same state of the maintained date. Thus, when reaching a selected join point the instance is added to the instance pointcut's multiset before (cf. line 6) any other action can access its current content. The other Attachments are created analogously. In the end, all Attachments are deployed through the ALIA4J System.

```
 1   public static void ${ipc}_deploy() {
 2     org.alia4j.fial.System.deploy(
 3       new Attachment(
 4         ${ipc}_add_before(),
 5         createStaticAction(void.class, ${Aspect}.class, ↙
                "${ipc}_addInstance", new Class[]{${Type}.class})
 6         SystemScheduleInfo.BEFORE_FARTHEST),
 7       Create Attachments for the other three parts analogously.
 8       For the ''after'' parts, use ↙
              SystemScheduleInfo.AFTER_FARTHEST.
 9       For the ''remove'' parts, specify method ↙
              ${ipc}_removeInstance.
10       );
11   }
```
Listing 17: Deployment of the bookkeeping for an instance pointcut.

To ensure this consistency also between AspectJ advice and our implementation of instance pointcuts, the AspectJ pointcut-advice definitions must be processed by ALIA4J. This provides an AspectJ integration which works together with the standard AspectJ tooling. Using command line arguments the AspectJ compiler can be instructed to omit the weaving phase. The advice bodies are converted to methods and pointcut expressions are attached to them using Java's annotations which are read by the ALIA4J-AspectJ integration and transformed into Attachments at program start-up. The code generated by our compiler consists of the above explained methods, as well as plain AspectJ definitions. When compiling this code with the mentioned command line options, the regular AspectJ pointcut-advice and the behavior of the instance pointcuts are both executed by ALIA4J, thus ensuring a consistent execution order.

4. RELATED WORK

AO-extensions for improving aspect-object relationships are proposed in several studies. Sakurai et al. [3] proposed Association Aspects. This is an extention to aspect instantiation mechanisms in AspectJ to declaratively associate an aspect to a tuple of objects. In this work the type of object tuples are declared with `perobjects` clause and the specific objects are selected by pointcuts. This work offers a method for defining relationships between objects. Similar to association aspects, Relationship Aspects [4] also offer a declarative mechanism to define relationships between objects, which are cross-cutting to the OO-implementation. This work focuses on managing relationships between associated objects. Bodden et al. [5] claim that the previous two lack generality and propose a tracematch-based approach. Although the semantics of the approaches are very similar, Bodden et al. combine features of thread safety, memory safety, per-association state and binding of primitive values or values of non-weavable classes. Our approach, also extending AO, differs from these approaches since our aim is not defining new relationships but using the existing structures as a base to group objects together for behavior extensions. Our approach also offers additional features of composition and refinement.

The "dflow" pointcut [6] is an extension to AspectJ that can be used to reason about the values bound by pointcut expressions. Thereby it can be specified that a pointcut only matches at a join point when the origin of the specified value from the context of this join point did or did not appear in the context of another, previous join point (also specified in terms of a pointcut expression). This construct is limited to restricting the applicability of pointcut expressions rather than reifying all objects that match certain criteria, as our approach does.

Another related field is Object Query Languages (OQL) which are used to query objects in an object-oriented program [7]. However OQLs do not support event based querying as presented in our approach. It is interesting to combine OQL like features with instance pointcuts; we will explore this in future work.

5. CONCLUSION AND FUTURE WORK

In this work we have presented instance pointcuts, a specialized pointcut mechanism for reifying categories of objects. Our approach provides a declarative syntax for defining events when an object starts or ends to belong to a category. Instance pointcuts maintain multisets providing a count for objects which participate in the same event more than once. Instance pointcut sets can be accessed easily and any changes to this set can also be monitored with the help of automatically created set monitoring pointcuts. The sets can be declaratively composed, refined to subsets or broadened to supersets, which allows reuse of existing instance pointcuts and consistency among corresponding multisets. Finally, we have presented a compilation approach for instance pointcuts based on the generation of plain AspectJ code.

The syntax and expressiveness of instance pointcuts partially depend on the underlying AO language; this is evident especially in our usage of the AspectJ pointcut language in the specification of events. Since AspectJ's join points are "regions in time" rather than events, we had to add the "before" and " after" keywords to our add and remove expressions. Thus, compiling to a different target language with native support for events (e.g., EScala [8] or Composition Filters [9], the point-in-time join point model [2]) would influence the notation of these expressions.

We think the instance pointcut concept is very flexible and can be useful in various applications. It eliminates boilerplate code to a great extent and provides a readable syntax. One relevant field of application for instance pointcuts are design patterns which are known to be good examples for aspect-oriented programming [10]. Many design patterns exist for defining the behavior for groups of objects; thus implementing them with our instance pointcuts seems to provide a natural benefit. We are currently working on creating object adapters wrapping the objects reified by an instance pointcut.

6. REFERENCES

[1] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An overview of aspectj. ECOOP 2001ŮObject-Oriented Programming (2001) 327–354

[2] Masuhara, H., Endoh, Y., Yonezawa, A.: A fine-grained join point model for more reusable aspects. In Kobayashi, N., ed.: Programming Languages and Systems. Volume 4279 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 131–147 10.1007/11924661_8.

[3] Sakurai, K., Masuhara, H., Ubayashi, N., Matuura, S., Komiya, S.: Design and implementation of an aspect instantiation mechanism. Transactions on aspect-oriented software development I (2006) 259–292

[4] Pearce, D., Noble, J.: Relationship aspects. In: Proceedings of the 5th international conference on Aspect-oriented software development, ACM (2006) 75–86

[5] Bodden, E., Shaikh, R., Hendren, L.: Relational aspects as tracematches. In: Proceedings of the 7th international conference on Aspect-oriented software development, ACM (2008) 84–95

[6] Kawauchi, K., Masuhara, H.: Dataflow Pointcut for Integrity Concerns. In Win, B.., Shah, V., Joosen, W., Bodkin, R., eds.: AOSDSEC: AOSD Technology for Application-Level Security. (March 2004)

[7] Cluet, S.: Designing oql: Allowing objects to be queried. Information systems **23**(5) (1998) 279–305

[8] Gasiunas, V., Satabin, L., Mezini, M., Núñez, A., Noyé, J.: Escala: modular event-driven object interactions in scala. In: Proceedings of the tenth international conference on Aspect-oriented software development. AOSD '11, New York, NY, USA, ACM (March 2011) 227–240

[9] Bergmans, L.M., Aksit, M.: How to deal with encapsulation in aspect-orientation. In: Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems. (2001)

[10] Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, ACM Press (2002) 161–173