Instance Pointcuts

An Aspect-Oriented Mechanism for Maintaining Object Sets

Kardelen Hatun, Christoph Bockisch, and 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 these phase shifts are typically implicit. In this study we propose a new language mechanism, called instance pointcuts, based on aspect-orientation. They maintain sets of objects according to events in their life-cycle and create notifications when new objects are added or removed from the set. 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 set operations. Our approach improves modularity by providing a fine-grained mechanism and a declarative syntax to maintain a set of objects.

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. And which role an object is currently playing is important as it can affect the object's own behavior or how objects are handled. Typically the shift from one life-cycle phase to another is implicit, e.g., determined by passing an object from one client to another. In this paper, we propose a new language mechanism for declaratively specifying life-cycle phases and for exposing the 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. Assume that the offered products are objects in the program and that specific budget plans should be offered for products depending on their life-cycle phase. A first example of such a phase is a period during which a product is in the wish-list of any customer; this phase begins when "product" object is added to the "wish-list" property of a "customer" object and it ends when it is removed from this property. A second example is the phase when an

object is out of stock. Thus, we defined two sets of objects: the set of objects into which at least one customer is interested and the set of objects that are sold out. Finally, the shop owner may be interested in the intersection of these sets and give priority to reordering the corresponding products.

Furthermore, in OOP objects are categorized according to their *types*. This is a structural categorization and it does not give any information about the events that object participates in. However, grouping objects according to another criteria such as, the class they were initialized in or being passed as an argument to a certain method would only be possible by inserting code at those particular points, which would litter the code.

To be able to process the objects which are currently in a relevant life-cycle phase (like having been added to a wish-list), bookkeeping is required to keep track of the set. To separate this bookkeeping code from the business logic of the program, aspect-oriented programming (AOP) is a well known technique. 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 offer dedicated mechanisms for selecting *sets of objects*.

These 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. In addition, a declarative implementation makes the relevant information explicit, which reduces checking efforts as well as making the code readable.

To offer better support for processing objects according to their life-cycle phase, we propose a language construct that builds on the technology of aspect-orientation. We borrow the terminology and provide *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 consists of two *pointcut expressions* that select relevant join points in the objects' life-cycle and expose the object: the mark the beginning and end of a life-cycle phase; i.e., the object is added to or removed from the set.

New instance pointcuts can be derived from existing ones. One instance pointcut can be declared to be a *subset* or a *super-set* of an existing one. In this case, the specification of the life-cycle phase is narrowed down or broadened, respectively. Composition of existing instance pointcut is also supported in terms of set operations: *intersection*, *union* and *set difference*.

2 Motivation - Example Section

Objects can be categorized by how they are used (passed as arguments to method calls, act as receiver or sender for method calls, etc.). Concerns may be applicable to objects used in the same way. Therefore we must be able to identify and select

those objects that are similarly used. Since object behavior is the building block of system behavior, such a categorization will allow behavior extensions on the object-level, providing an extra dimension of modularity.

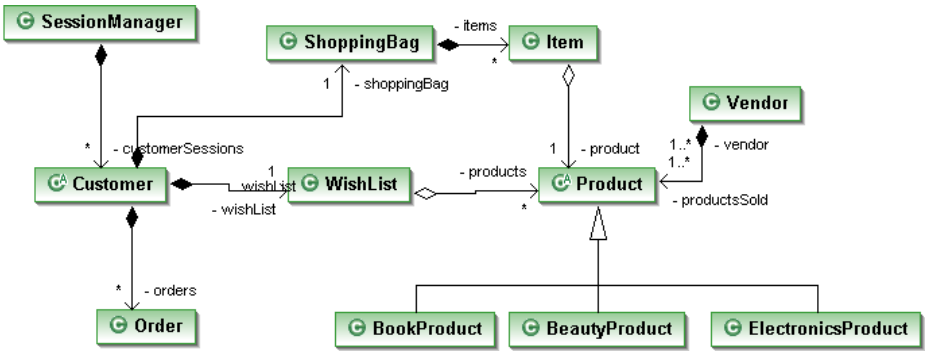


Fig. 1. A simple online shop application

In the remainder of this section we outline the architecture and design of an online store application. Then 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 unanticipated concerns. Finally we conclude requirements for solving the encountered challenges in these examples.

Online Shop An online shop is a sophisticated web application and objects of the same type can exist at different stages of the control flow. 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 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, an new `Item` instance is created and added to the `ShoppingBag` object; the `Item` instance contains the `Product` and how many of that `Product` was added. A customer can add/remove items from his shopping bag. When the shopping is finished then the `checkOut()` method is invoked, which returns an object of type `Order`. To complete the order the customer has to provide some information and finalize the order. A `Customer` can olso wishlist `Products` by adding them to his `WishList`; the `WishList` holds a list of `Products` (whereas `ShoopingBag` holds a list of `Items`).

Let's assume a new requirement for applying a happy-hour discount is introduced. The discount should apply to a set of `Items` which are selected when they exist in the context of an interesting event. The new requirement is as follows, when a customer adds items to his shopping bag between certain hours, then

```
1  Set<Item> happyItems = createSet();
2  public boolean addToShoppingBag(Product p, int amount)
3  {
4    Item item = ItemFactory.createItem(p,amount);
5
6    if('timing condition')
7      items.put(item)
8
9    this.shoppingBag.add(item);
10 }
11 public boolean removeFromShoppingBag(Item item, int amount)
12 {
13   ...
14   items.remove(item);
15 }
16 public checkOut()
17 {
18   foreach(i in happyItems)
19     ProductManager.applyDiscount(i, DiscountRule.get(HAPPY_HOUR)
         );
20 }
```

Listing 1. A Java implementation of Happy-hour discount rule

those items will be applied a discount in check out. In order to realize this in an OO-approach, one needs to insert the code regarding the discount concern, to the place where the event of interest occurs. With an OO-approach, to satisfy the new requirement we need to invasively change the system code. First we need to keep a list of items that were added to shopping bag inside the timing condition. Then when the user finally checks out, we should apply the discount to the items in the list. 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), to keep track of items that are added during the happy-hour. In `addToShoppingBag` method a if statement checking the timing condition is defined, when the condition is true than the created item is added to `happyItems` (Line 6-7). Finally in the `checkOut` method the discounts are applied to the items in `happyItems` (Line 18-19).

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 way of implementing is clearly not suitable. Also this event may not be the only event we want detect for applying discount rules and the OO implementation will eventually be scattered and tangled. An aspect-oriented implementation can offer a better solution, by encapsulating the concern in an aspect. In Listing 2 the `newItem` pointcut (Line 2-4) selects join-points where after customer adds a product to his bag, a new `Item` is created in `addtoShoppingBag` method. The advice is executed when `newItem` is matched; after returning the new item, the timing condition

```
1  Set<Item> happyItems = createSet();
2  pointcut newItem(Item item):
3      call(* ItemFactory.createItem(..)) &&
4      withincode(* Customer.addToShoppingBag(..));
5  pointcut removeItem(Item item):
6    call(* Customer.removeFromShoppingBag(..)) &&
7    args(item);
8  pointcut checkOut(): call(* Customer.checkOut());

10 after() returning(Item item): newItem(item)
11 {
12   if('timing condition')
13     happyItems.put(item);
14 }
15 after(Item item): removeItem(item)
16 {
17   happyItems.remove(item);
18 }
19 before():checkOut()
20 {
21   foreach(i in happyItems)
22     ProductManager.applyDiscount(i, DiscountRule.get(HAPPY_HOUR)
         );
23 }
```

Listing 2. An Aspectj implementation of Happy-hour discount rule

is checked and if it hols than the item is added to `happyItems`. We also define a `removeItem` pointcut (Line 5-7) to remove an item from `happyItems` if it is removed from shopping bag. Finally, we define a pointcut for selecting the `Customer.checkOut()` join-point (Line 8) and before the method is executed we apply the discount to items in `happyItems`. AO implementation, although offers a non-invasive solution, also suffers from boilerplate code written for book-keeping.

Let's look at another new requirement, which is as follows: Keep a set of products that are wishlisted and currently out of stock, once they become available raise an event and remove them from the set. This scenario also requires when a condition holds and the set is changed there should be an event raised which means the set should be monitored. This creates an additional concern.

Reusing existing sets by finding their subsets or composing them with set operations is also impractical with imperative programming. If we want to find the subset of `BeautyProducts` in a set of `Product` object, we have to iterate over it and check instance types to create a new set. Implementing set operations such as intersection also results in further boilerplate code.

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 scenarios described above, we conclude the following requirements:

Requirement1: Provide a declarative way of reifying a set of objects by defining add/remove conditions
Requirement2: Support multi-sets, an object can be added multiple times
Requirement3: Add/Remove expressions select events and the object is exposed from the context of this event
Requirement4: Access the set of objects which is currently selected and be notified when the set changes
Requirement5: Provide a declarative way to refine existing sets to their subsets
Requirement6: Compose sets via set operations (intersection, union, set difference)
Requirement7: Provide checks for validity of composition, empty sets and subset-superset compatibility

3 Instance Pointcuts

Instance pointcut is a declarative language construct that is used to reify and maintain a set of objects of a specific type, with the ability to select them over a period marked by events in their life-cycle, modularizing the object selection concern. The purpose of instance pointcuts is to let the user to make focused selections, therefore offering a different level of modularity.

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 in AspectJ.

maybe section outline summarizing main features

3.1 Features

A concrete instance pointcut definition consists of a left hand-side and a right-hand side. In the left hand side the pointcut's name and the instance type of interest is declared. Instance pointcuts do not declare pointcut parameters, it has a single implicit parameter called `instance` of the declared instance type. In the right hand side a pointcut expression describes the desired join-points and then binds the exposed object as a member of the instance pointcut's set, which is represented by the variable `instance`. It is also possible to declare an abstract instance pointcut, by leaving out the right hand side and placing the *abstract* modifier at the beginning of the declaration and not defining a pointcut expression. Abstract instance pointcuts are used when execution details are not yet clear but the instance set to be created is determined.

Add/Remove Expressions Join points mark *sites* of execution; a join-point by itself does not define an event. In AspectJ pointcuts, pointcut expressions select joinpoints and pointcuts are used with advice specifications to select a particular event in that join-point. Instance pointcuts are not used with advices; since they serve the specific purpose of reifying and maintaining object sets, they

implicitly perform add to set and remove from set operations. Therefore instance pointcut expressions are used to select specific events rather than joinpoints. To be able to define events with instance pointcut expressions, it is necessary to use advice specification clauses `before` or `after` in the expression.

The *add expression* select events which contain the instances to be added to the instance pointcut set; an event selected by add expression marks the beginning of the life-cycle phase of interest. Instance pointcuts can also include an optional `remove expression`, which specifies when to remove which object, selects the event that marks the end of the life-cycle phase of interest. 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*. Each element contains a pointcut expression and they can select more than one event. The before element selects the before events and the after element, selects after events.

The grammar definition below shows part of the instance pointcut syntax. An event definition is comprised of an event keyword , `before` or `after`, and a pointcut expression. The pointcut expression selects a set of join points and the event keyword specifies which event is selected from a join-point. 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.

⟨*instance pointcut*⟩ ::= ‘`instace pointcut`’ ⟨*name*⟩ ‘<’ ⟨*instance-type*⟩ ‘>’ ‘=’
⟨*ip-expr*⟩ (‘`UNTIL`’ ⟨*ip-expr*⟩)?

⟨*ip-expr*⟩ ::= ⟨*after-event*⟩ ‘`||`’ ⟨*before-event*⟩
| ⟨*before-event*⟩ ‘`||`’ ⟨*after-event*⟩
| ⟨*after-event*⟩
| ⟨*before-event*⟩

⟨*after-event*⟩ ::= ‘`after`’ ‘`(`’⟨*pointcut-expression*⟩‘`)`’

⟨*before-event*⟩ ::= ‘`before`’ ‘`(`’⟨*pointcut-expression*⟩‘`)`’

Fig. 2. Grammar definition for instance pointcuts

The instance pointcut in Listing 3 shows a basic example. The left-hand side of the instance pointcut indicates the pointcut is called `customers` and it is interested in selecting the `Customer` objects. On the right hand side, there are two expression separated by the `UNTIL` keyword. The first one is the add expression. It selects the return event of the method `createCustomer` and

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

Listing 3. A basic instance pointcut declaration with add and remove expressions

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 bounded 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. Because 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 `customers` pointcut, if the `Customer` instances were collected by the garbage collector, they would still be removed from the set.

Multiset Support Instance pointcuts provide the option to use sets or multisets. Sets are restricted since they can only contain an element once, multisets how ever can contain duplicates of an element. By default the following expression **`instance pointcut`** `{pc_name}<{instance_type}>` means that the instance pointcut is a set. The multiset option can be chosen via putting `*` after instance type; **`instance pointcut`** `{pc_name}<{instance_type*}>`.

Multisets are used when the same object is added to the set multiple times and there should be book-keeping of the duplicates. The examples in Listing 4 shows two situations one using a set and another using a multiset. The add expression of `products1` selects the products which are added to a customer's wish-list and which are currently out of stock, until a `Vendor` stocks up that product again then the product is removed from the set. In this case we use a set because we are not interested in the duplicates of `Product` instances, we are only interested if a `Product` is wish-listed when it was out of stock. The second instance pointcut also selects `Product` instances, this time without checking the availability. 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 count the currently wish-listed products. If we use a set then as soon as a customer removes a product from his wish-list, its only copy will be removed and it will appear removed from all customers' wish-lists, therefore a multi-set is used.

Set Monitoring An instance pointcuts definition also defines two set change events; add event and remove event. These events are interesting since they signify that the object has become relevant or irrelevant. In order to select the join-points of these events, every instance pointcut definition automatically has two implicit regular pointcuts. These implicit pointcuts has the following naming convention; `<name>_add, <name>_remove`, where `<name>` is the name of the

```
instance pointcut products1<Product>:
before(call(Customer.addToWishList(..)) && args(instance) && if
    (!instance.inStock()))
  UNTIL after(call(Vendor.stock(..)) && args(instance))
instance pointcut products2<Product*>:
before(call(Customer.addToWishList(..)) && args(instance))
 UNTIL before(call(Customer.removeFromWishList(..)) && args(
     instance))
```

Listing 4. Two instance pointcuts, each maintains a set of wish-listed products

instance pointcut. These pointcuts allow the user to, for example raise an event when an object becomes relevant. If we remember the following requirement form Section 2; Keep a set of products that are wishlisted and currently out of stock, once they become available raise an event and remove them from the set. The instance pointcut defined in Listing 4 ad `products1` almost satisfies this requirement. To fully satisfy the requirement we can use the implicit pointcut `products1_remove` as in Listing 5. This advice declaration indicates an event will be raised before the product is removed from the set. It is also possible to

```
before(): products1_remove()
{
  raiseEvent();
}
```

Listing 5. Implicit remove pointcut used with an advice

use implicit pointcuts with parameter binding, as shown in Listing 6; the advice daclaration is the same, but this time removed instance is exposed and passed to the `raiseEvent` method.

```
before(Product p): products1_remove(p)
{
  raiseEvent(p);
}
```

Listing 6. Implicit remove pointcut with parameter binding used with an advice

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.

Referencing and Dynamic Refinement Normally instance pointcuts are referenced by their name, however they can also take an additional statement

for *type refinement*, which selects a subset of the instance pointcut *dynamically*. Type refinements require that, the refined type is a subtype of the original instance type. For example the instance pointcut `lowSelling` (Listing 8) can be refined with the following syntax: `lowSelling<BeautyProduct>`. This indicates we would like to select the subset of `BeautyProduct` instances from the set of `Product` instances selected by `lowSelling`. Note that this notation will also select subtypes of `BeautyProduct`. The effect of a refinement subset being empty is equivalent to that of an unmatched pointcut.

Instance Pointcut Composition It is also possible to compose instance pointcuts to obtain a more refined or a broader set. In Figure 3, an extended version of the grammar definition in Figure 2 is shown. A new pointcut can be defined either by referencing another pointcut and appending additional predicates to the pointcut expressions in before and after blocks or it can be defined as the composition of other pointcuts or both. In order to obtain a more refined set && operation is used, for a broader set ‖ is used.

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

⟨*logical-operator*⟩ ::= '`&&`'
| '`||`'

⟨*ip-ref-expr*⟩ ::= ⟨*ip-ref*⟩
| ⟨*ip-ref-expr*⟩ '`&&`' ⟨*ip-ref-expr*⟩
| ⟨*ip-ref-expr*⟩ '`||`' ⟨*ip-ref-expr*⟩

⟨*ip-ref*⟩ = ⟨*name*⟩
| ⟨*name*⟩'<' ⟨*refined-instance-type*⟩ '>'

Fig. 3. Extended grammar definition for instance pointcuts

An instance pointcut can have at most four expression elements; before (part of) 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 following notation $ba_{IP}\_expression$.

$$IP1 = ba_{IP1} \parallel aa_{IP1} \text{ UNTIL } br_{IP1} \parallel ar_{IP1} \quad (1)$$

$$IP2 = ba_{IP2} \parallel aa_{IP2} \text{ UNTIL } br_{IP2} \parallel ar_{IP2} \quad (2)$$

Let us define a new pointcut as; $IP3 = IP1$ && $IP2$. This operation is results in the following formulation:

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

$$\begin{aligned} IP3 \rightarrow \text{remove expression} = &(br_{IP1} \parallel br_{IP2}) \\ &\parallel (br_{IP1} \parallel ar_{IP2}) \\ &\parallel (ar_{IP1} \parallel ar_{IP2}) \\ &\parallel (ar_{IP1} \parallel br_{IP2}) \end{aligned} \tag{4}$$

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:

$$\begin{aligned} event(exp_1) \;\&\&\; event(exp_2) &= event(exp_1 \;\&\&\; exp_2) \\ event(exp_1) \parallel event(exp_2) &= event(exp_1 \parallel exp_2) \end{aligned}$$

Note that we do not evaluate 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.

Then IP3 becomes;

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

Notice that the remove expressions are applied $\parallel$ instead of &&. If we narrow down 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 pointcuts, we ensure that all the added object can be removed. So we can say that the && and the $\parallel$ 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:

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

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

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

Let us illustrate these concepts with a concrete example. The first instance pointcut shown in Listing 8 selects the `Products` which are marked as *low-selling* by the `OnlineShop` class, its add expression contains a single after event. The second pointcut selects the `Products` which are added to the wish-list of a `Customer`, and also contains a single after event. Then the add expression of `composition` is created by factoring out the `args` pointcut as; **`after`**`((`**`call`**`(* OnlineShop.setLowSelling(..))||` **`call`**`(* Customer.addToWishList(..)))&&` **`args`**`(`**`instance`**`))`.

```
1 instance pointcut lowSelling<Product>: after(call(* OnlineShop.
    setLowSelling(..))  && args(instance));
2 instance pointcut wishList<Product>: after(call(* Customer.
    addToWishList(..)) && args(instance)) ;
3 instance pointcut composition<Product>: lowSelling || purchase;
```

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

Similarly we can define a new pointcut, this time by composing a pointcut with an expression element, then the `composition2` pointcut has the following add expression; **`after`**`(`**`call`**`(* OnlineShop.setLowSelling(..))&&` **`args`**`(`**`instance`**`)&&` **`if`**`(`**`instance`**`.getPrice()> 10))`.

```
instance pointcut composition2<Product>: lowSelling && after(if(
    instance.getPrice() > 10));
```

Listing 8. Two instance pointcuts composed to obtain a new one

Subset and Superset Relationships There are two mechanism to define relationships between instance pointcuts, *subsetting* and *supersetting*. The declaration of these relationships impose some restrictions on the *declaring* instance pointcut. Let us update the syntax definition to host relationship declarations.

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

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

Fig. 4. Grammar definition for relationship declarations

For the following declaration `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`. Hence, if later in the implementation, the scope of `subsetIP1` is broadened, it should still follow the subset restriction. When the*instance type* is not declared like in the example then it is inherited from the superset `IP1`. It is also possible to declare subsets by using type refinement. For example `subsetIP1<SubType>` **`subsetof`** `IP1 = IP1<SubType>`, creates a subset of `IP1` by selecting the instances of type `SubType`. For the superset declaration `supersetIP1` **`supersetof`** `IP1`, the superset relationship imposes that the instance set created by `supersetIP1` has to contain the set of `IP1`. This rule should hold even if the scope of `supersetIP1` is narrowed. When the instance type is not declared then the instance type of `IP1` is copied.

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

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 pointcuts, which may result in erroneous selections. Since the subsets only supports the subtypes of the original instance type and superset support only the supertypes, it works with the system's type hierarchy in a natural manner.

example

3.3 Compilation of Instance Pointcuts

Instance pointcuts are implemented as an extension to AspectJ, they reuse AspectJ pointcut expressions, however specific use of advice specifiers `before` and `after` allow use of `returning` in the pointcut expression to capture returned objects. They are also compiled to AspectJ code, in this section we will discuss the compilation steps and which structures an instance pointcut is mapped to.

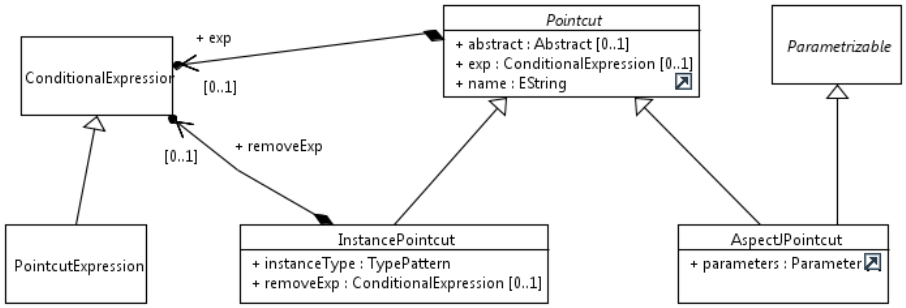
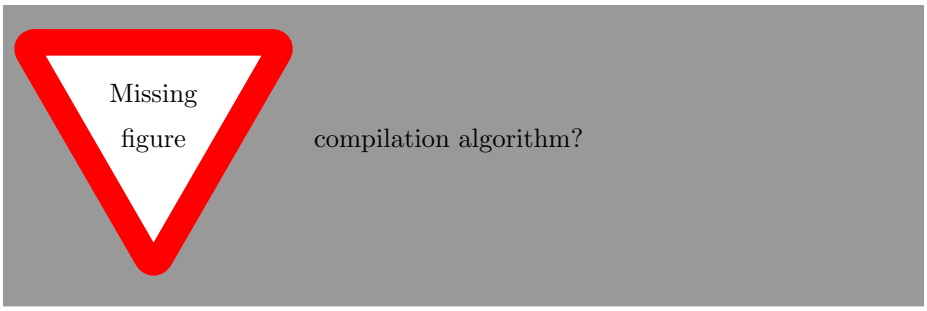


Fig. 5. Instance pointcut static structure

4 Compilation of Instance Pointcuts

AspectJ code generation



5 Related Work

6 Discussion

7 Conclusion and Future Work

keep it short