

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.

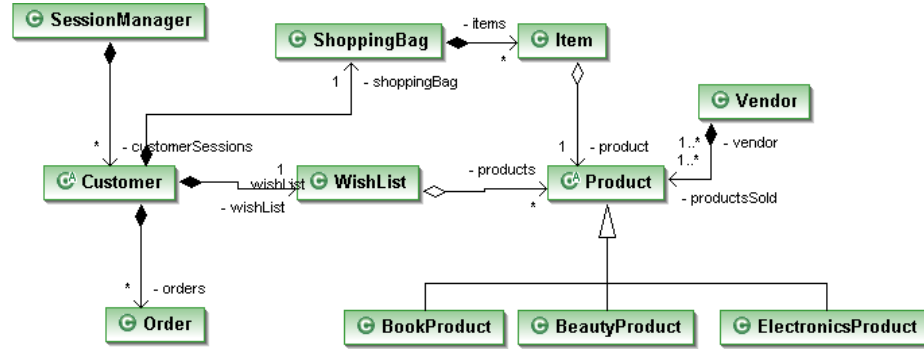


Fig. 1. A simple online shop application

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 also wishlist Products by adding them to his WishList; the WishList holds a list of Products (whereas ShoppingBag 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

Listing 1.1. A Java implementation of Happy-hour discount rule

```

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             );
21 }

```

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 1.1 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 then 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 1.2 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

Listing 1.2. An Aspectj implementation of Happy-hour discount rule

```

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             );
24 }

```

when `newItem` is matched; after returning the new item, the timing condition is checked and if it holds then 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. Unlike AspectJ pointcuts, instance pointcuts cannot declare pointcut parameters, it only 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.

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 have advice bodies. 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* which exposes the instances to be added to the instance pointcut set, selects the event which marks the beginning of the life-cycle phase of interest. Instance pointcuts can also include an optional *remove expression*, which exposes the instances to be removed from the set, selects the event which marks the end of the life-cycle phase of interest. Both *add* and *remove expressions* contains at least one *event* definition at at most two; the *before event* and *after event*. 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 reason for this is, the AND operation will result in an empty set, since a *before* and an *after events* are separate.

```

<instance pointcut> ::= 'instance pointcut' <name> '<' <instanceType> '>' '='
    <ip-expr> ('UNTIL' <ip-expr>)?

<ip-expr> ::= <after-event> '|' <before-event>
    | <after-event>
    | <before-event>

<after-event> ::= 'after' '(' <pointcut-expression> ')'
<before-event> ::= 'before' '(' <pointcut-expression> ')'

```

Listing 1.3. A basic instance pointcut declaration with add and remove expressions

```

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

```

The instance pointcut in Listing 1.3 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 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 however 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 important when the same object is added to the set multiple times. The examples in Listing 1.4 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. 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, it will appear removed from all customers' wish-lists, therefore a multi-set is used.

Listing 1.4. Two instance pointcuts, each maintains a set of wish-listed products

```

1 instance pointcut products1<Product>:
2 after(call(Customer.addToWishList(..)) && args(instance) && if(!
   instance.inStock()))
3 UNTIL after(call(Vendor.stock(..)) && args(instance))
4 instance pointcut products2<Product*>:
5 after(call(Customer.addToWishList(..)) && args(instance))
6 UNTIL after(call(Customer.removeFromWishList(..)) && args(
   instance))

```


How to use them and invalid combinations

Additional checks?



Example using before after statements

3.2 Instance Pointcut References

how to reference instance pointcuts

Type-refined references

3.3 Instance Pointcut Composition

3.4 Subsets

Explain what type of relationships subsets stand for

Paragraph: subsetting by type refinement

Paragraph: subsetting by narrowing down the pointcut expression with additional predicates



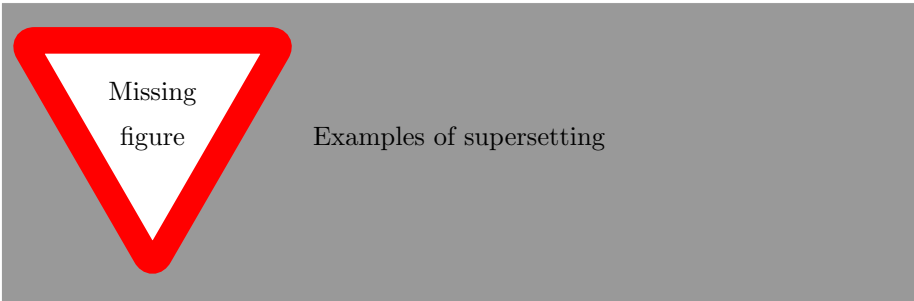
Examples of subsetting, type refinement, additional predicates and both

3.5 Supersets

Explain what type of relationships supersets stand for

Paragraph: superset by composing multiple instance pointcuts refer to union operation

Paragraph: superset by composing an instance pointcut with an expression



3.6 Union, Intersection and Set Difference

Define these operations and give simple examples

3.7 Refinement

It is possible to refine instance pointcuts in multiple ways. 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 `customers` can be refined with the following syntax: `customers<GoldCustomer>`. This indicates we would like to select the subset of `GoldCustomer` instances from the set of `Customer` instances selected by `customers`. Note that this notation will also select subtypes of `GoldCustomer`. If this is not the desired effect then the following: **instance pointcut** `goldCustomers<GoldCustomer>`: `customers` && **if**(`instance.getClass().equals(GoldCustomer.class)`) can be used. The effect of a refinement subset being empty is equivalent to that of an unmatched pointcut.

Instance pointcuts can also be refined inside a pointcut expressions and the refinement result will then be assigned to the instance pointcut on the left-hand side. Once again refining the `customers` pointcut, we can define the following:

```
1 instance pointcut goldCustomers<GoldCustomer>: customers && if(
    instance.getGoldCustomerID().startsWith('NL'))
```

The `goldCustomers` instance pointcut is interested in `GoldCustomer` objects which are selected by `customers` and who have `goldCustomerIDs`

starting with the string “NL”. The pointcut expression to select these instances is very concise and is shown in the listing above. `customers<GoldCustomer>` statement selects the `GoldCustomer` instances selected by `customers` pointcut, then the `if` pointcuts checks whether these instances satisfy the condition. Note that the following statement; **instance pointcut** `regularCustomers<RegularCustomer>: customers<GoldCustomer> && if(instance.getGoldCustomerID().startsWith('\nL'))`, would result in a compile error for two reasons. First we cannot assign `GoldCustomer` instances to a `RegularCustomer` instance pointcut. Secondly `instance.getGoldCustomerID()` is illegal since `instance` is the implicit variable of the `regularCustomers` pointcut and it has the type `RegularCustomer`, which does not have the property `goldCustomerID`.

Refinement mechanisms provide a consistent selection process and reduce redundancy. Being able to select subsets of instance pointcuts, eliminates the need for defining separate pointcuts for subsets, which may result in erroneous selections. Since the refinement only supports the subtypes of the original instance type, it works with the system’s type hierarchy in a natural manner.

3.8 Compilation of Instance Pointcuts

Instance pointcuts are implemented as an extension to AspectJ, they reuse AspectJ pointcut expressions, however they also allow returning clause in the pointcut expression to capture returned objects. In Figure 3.8 static structure of this extension is shown. Both `InstancePointcut` and `AspectJPointcut` inherit from the `Pointcut` abstract class. As seen on the diagram, instance pointcuts can have two pointcut expressions, one is inherited from `Pointcut` and is called `exp`. This attribute is also inherited by the AspectJ pointcut. The second one is however specific to instance pointcuts (`removeExp`) and it can be used to define pointcut expressions for deselecting instances(Section ??). Note that the `PointcutExpression` class is a subtype of `ConditionalExpression` (Mention JaMoPP). Another notable difference between AspectJ and instance pointcuts is that AspectJ pointcuts are parametrizable (`Parametrizable` interface), while instance pointcuts have a single implicit parameter called *instance*. The type of this parameter is stored in the mandatory attribute `instanceType`.

Instance pointcuts can be compiled to various AO-languages, for our prototype we chose AspectJ as the target language. A pseudo-code of the generator template for instance pointcuts is shown in Listing ??.

4 Compilation of Instance Pointcuts

AspectJ code generation

Generation of instance pointcut related hooks in the form of aspectj pointcuts, such as add/remove operations to the instance pointcut set

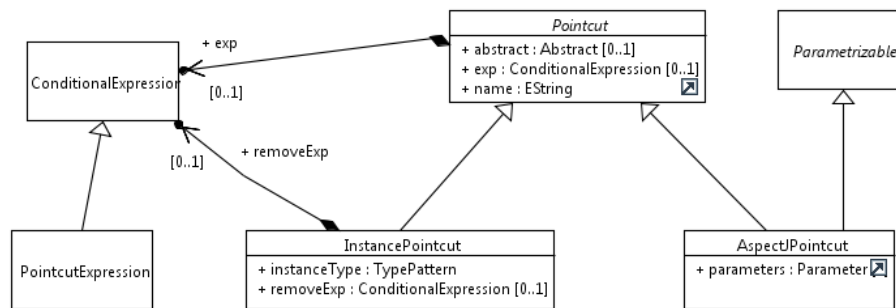
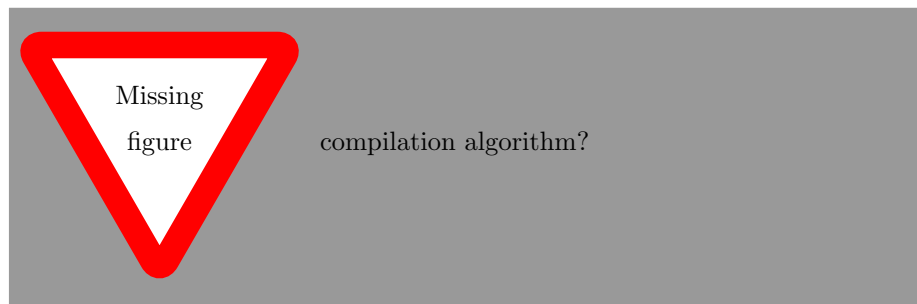


Fig. 2. Instance pointcut static structure



5 Related Work

6 Discussion

7 Conclusion and Future Work

keep it short