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.

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 some concerns. Finally we conclude requirements from 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. Typically such objects have properties which indicate their *state*. At certain points in execution these states are updated accordingly. 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` which may contain arbitrary *items* and a `WishList` which may contain arbitrary *products* . The class `Item` represents a concrete `Product`, e.g. when the user selects a product and clicks add to shopping bag, an `Item` is added to the `ShoppingBag` object, aggregating the selected `Product`. The abstract class `Product` is super-type of all products in the shop. The `Product` is further broken down to product categories such as beauty, electronics etc. There are two customer types in the system, `RegularCustomer` and a `GoldCustomer`. `GoldCustomers` can benefit from additional discounts when buying certain products. A customer can add/remove items from his shopping bag. When the shopping is finished then the `purchase()` method is invoked which returns an object of type `Order`. To complete the order the customer has to provide payment information (`PaymentInfo`) and shipping information(`ShippingInfo`). Once this information is complete the order is finalized.

A New Requirement Let's assume a new requirement for applying more sophisticated discount rules to products is introduced. The rules should should apply to a subset of `Item`s which are selected when they exist in the context of an interesting event. 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. For the following discount rule; when an item is added to a shopping bag between certain hours, then they are applied the happy-hour discount, we need to invasively change the system code by inserting a check for *timing condition* and a method call for applying the discount rule should the condition hold. In Listing 1.1 the code is inserted in the method `addtoShoppingBag` of `Customer` class.

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

```
1 public boolean addToShoppingBag(Product p, int amount)
2 {
3   Item item = ItemFactory.createItem(p,amount);
4   /*The discount concern*/
5   if('timing condition')
```

```
6       ProductManager.applyDiscount(item, DiscountRule.getRule(
            HAPPY_HOUR));
7     /*******************/
8     this.shoppingBag.add(item);
9   }
```

Even for a single discount rule, the code for the discount concern creates cluttering. If we would like to apply multiple discount rules for the *add to shopping bag* event, this way of implementing is clearly not suitable. Maybe another solution can be changing the method signature of `addtoShoppingBag` to include a parameter for `DiscountRule`. However 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 `pc` pointcut selects join-points where customer adds a product to his bag. The advice is executed when `pc` is matched; after the timing condition is checked the desired discount is applied.

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

```
1  pointcut pc(Item item): call(* ItemFactory.createItem(..)) &&
       withincode(* Customer.addToShoppingBag(..));
2
3  after() returning(Item item): pc(item)
4  {
5    if('timing condition')
6      ProductManager.applyDiscount(item,DiscountRule.getRule(
           HAPPY_HOUR));
7  }
```

If we make the same discount rule more complex by saying; apply a happy-hour discount only to beauty products then we should insert an *instance of* check to the `Product` the `Item` instance holds, to apply the right kind of discount. This situation becomes more complex when there are multiple discount rules, for various types of products. This imperative way of selecting objects becomes error-prone. The aspect implementation, while successfully encapsulating the discount concern, becomes riddled with boilerplate code and it suffers from the same error-prone checks the OO implementation does. Once the happy hour is over, the discount needs to be removed from items. With the current implementation, we do not know which items are applied a discount, because we did not do any book-keeping. In order to remove the discounts, we either have to do a complicated checking regarding the type of the products and when it was added to the shopping bag, or there should be a global structure holding these set of elements, which means adding book-keeping concern to the implementation.

Instead of creating a new set structure everytime a discount rule is applied, we may want to select a subset of an existing set to apply the discount. For example in a set of `BeautyProduct`s, we may want to access the subset `HandCream`s. Java requires we iterate over the set and check each product's type to see if it is of

type `HandCream`. With AspectJ, we can select join-points where an element is put to the beauty products set, and expose the element to check if it is an hand cream. Both of these imperative implementations suffer from boilerplate code and reusing the sets for refinement or other set operations causes makes the code unreadable and error-prone.

what needs to change to support them?

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: reifying set of objects, adding/removing object from the set
Requirement2: multi-set
Requirement3: Events and select objects from the context of the event
Requirement4: access the set of objects which is currently selected and be notified when the set changes
Requirement5: refinement: subset and support
Requirement6: composition: set operations
Requirement7: checks: validity of composition, empty sets

2.1 Explanation of features, just a summary

Selecting a set of objects based on their life-cycle events provide a new dimension in categorizing objects. Instance pointcuts reify such a set, by offering a declarative syntax for adding objects to it. The object to be added is exposed in the context of the matched event. An object might only be relevant during a particular phase of its life-cycle, which is marked by certain events. Then it is necessary to have a mechanism to remove objects from the set once the event that marks the end of that phase is encountered. Hence, it is possible to declare an optional removal expression in an instance pointcut.

Instance pointcuts represent a set, and the objects in this set can be accessed through instance pointcuts. Also when there's a change in the set, i.e. an object is added or removed, it is possible for the interested modules to access the change events.

Sets can have subsets or supersets. This is also true for the sets of objects created by instance pointcuts. An instance pointcut can be refined to obtain the subset of objects it originally selected or it can be a subset of another instance pointcut which broadens its scope. This refinement mechanism makes instance pointcuts reusable. Composition is also supported via the ability to perform set operations on instance pointcuts. It is possible to define intersection or union of two instance pointcuts, in order to define a new one.

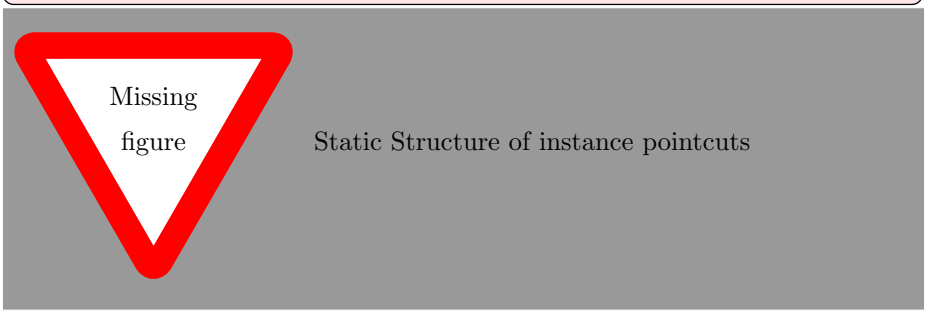
The various features instance pointcuts offer also require some checking to be performed. For example when refining a pointcut to obtain a subset, we may end up with an empty set. This situation should be checked and necessary warning should be given. Since instance pointcuts dedicated to a type, the composition

operations may fail if the composed pointcuts do not select co-variant types, which should also be checked.

3 Instance Pointcuts

Instance pointcut is a declarative language construct that is used to select a set of objects of a specific type. The result of the selection is a *set*, hence it cannot contain the same object more than once. Instance pointcuts provide the ability to select objects over a period marked by events in their life-cycle, modularizing the object selection concern. It also provides a mechanism to construct a set of objects according to relevant events, the places in application those events take place and object state at a certain point in execution. Instance pointcuts lets the user to make focused selections,therefore manage the system at a finer level.In the remainder of this section, we will explain instance pointcuts in detail.

maybe section outline summarizing main features



Static Structure of instance pointcuts

3.1 Basic Syntax

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 cannot declare variables, it only has a single implicit variable 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.

```
1 instance pointcut customers<Customer>: call(SessionManager.
      createCustomer(..)) && returning(instance)
```

The instance pointcut above 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, the pointcut expression selects the join-points where `Customer` is returned by the `createCustomer` of the class `SessionManager`. The returned instances are exposed and bound by the `returning` clause to the *implicit variable* `instance`, which represents a member of `customers`' set. After selecting a set of objects, instance pointcuts offer ways to manipulate this set.

3.2 Supported AspectJ Predicates

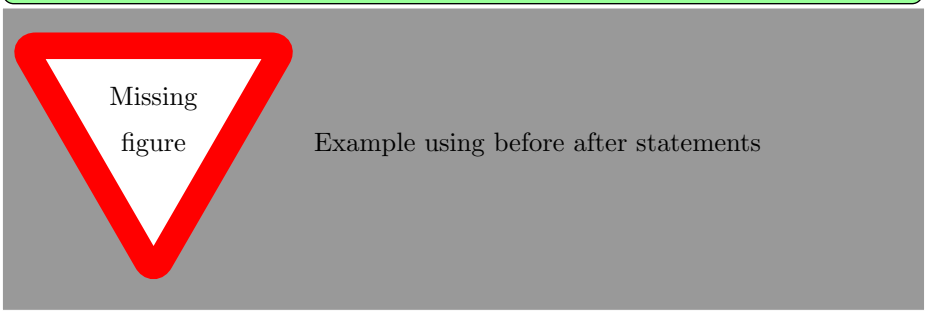
Here we will discuss which predicates we support, with additional explanation of the returning clause

3.3 before / after Statements

How to use them and invalid combinations

What is the default behavior if they are not explicitly defined

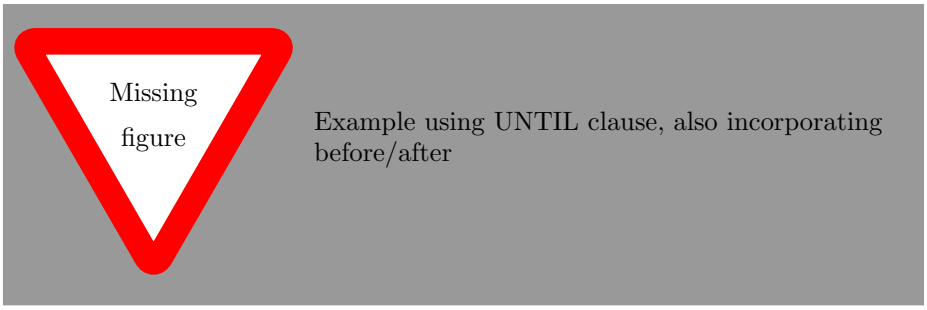
Additional checks?



Example using before after statements

3.4 Deselection

How to deselect instances, what are possible checks?



Example using UNTIL clause, also incorporating before/after

3.5 Instance Pointcut References

how to reference instance pointcuts

Type-refined references

3.6 Instance Pointcut Composition

3.7 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

3.8 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

Examples of supersetting

3.9 Union, Intersection and Set Difference

Define these operations and give simple examples

3.10 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 `goldCustomerID`s 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.

3.11 Deselecting Instances

The examples of instance pointcuts presented so far were for selecting objects. Once an object is selected, it does not have to remain in the instance set until it dies. It is possible to define removal conditions in the form of pointcut expressions, that can point to any event in the object's life-cycle.

In the listing below the instance pointcut `rectangle` selects the subset of `Rectangle` objects which are selected by `shapes` pointcut. These instances are selected *until* the pointcut expression that follows the keyword `UNTIL` is true.

```
1 instance pointcut rectangle<Rectangle>: shapes<Rectangle>
    UNTIL call(* Rectangle.setWidth(..)) && if(instance.
    getWidth() > 10) && target(instance)
```

The ability to deselect instances provides flexibility over managing instances. With this mechanism user can select a period in an instance's life-cycle where the beginning and the end of the period is marked by pointcut expressions.

Advice before / after?

3.12 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 3.12 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 3.11). 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

compilation algorithm?

5 Related Work

6 Discussion

7 Conclusion and Future Work

keep it short

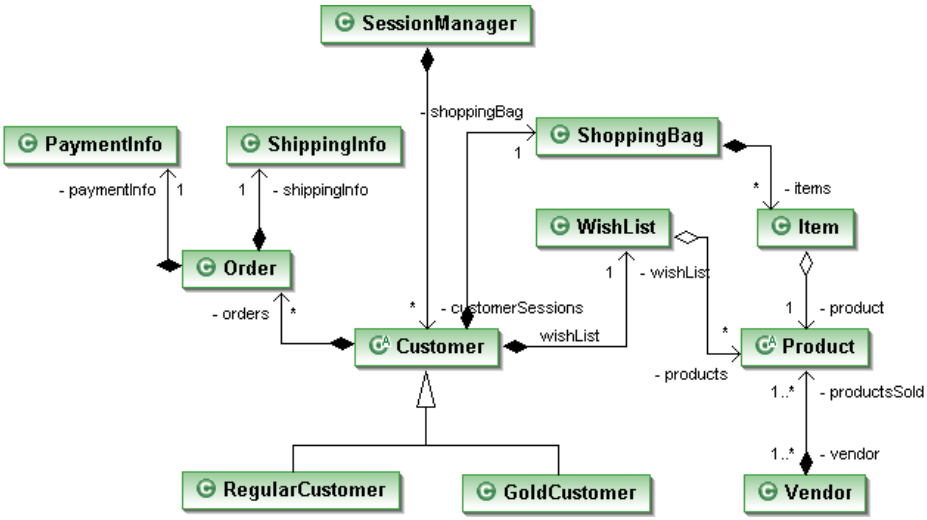


Fig. 1. A simple online shop application

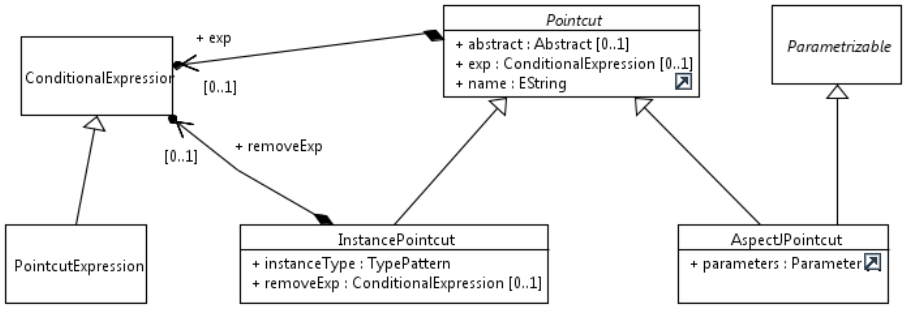


Fig. 2. Instance pointcut static structure