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., by restricting the scope 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.

Categories and Subject Descriptors

D.3.1 [**Formal Definition and Theory**]: [syntax, semantics]; D.3.4 [**Processors**]: [code generation]

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 than 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 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 based on how they are used by customers: 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 that is not directly supported by the programming language, such as which class they were initialized in, which method they were 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 phase; 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 a new mechanism, called *instance pointcuts*, to select sets of objects based on the events in their execution history. We present our prototype which is implemented as an extension to AspectJ. 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.

Firstly, a new instance pointcut can be derived from another one by restricting the type of selected objects. Secondly, 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 explaining our compiler which transforms instance pointcuts to plain AspectJ and advanced dispatching library calls.

The rest of the paper is organized as follows, in section 2 we present a small case study and explain our motivation for the proposed approach. In section 3 a detailed description of instance pointcuts and its various features are presented. This section also explains how instance pointcuts are compiled. We then present a discussion on the validation of our approach. We conclude by discussing related work and a summary of our approach.

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 phase-dependent concerns can explicitly refer to the category.

In Figure 1, we outline a part of the architecture of an online store application from the *system's perspective*. 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 their life-cycle. In Figure 1 the static structure of a simplified online shop is shown. This structure shows part of the system from the Vendor and the OnlineShop's perspective. Vendors can submit different kinds of Discounts to the ProductManager for the Products they are selling. `Product` is the root of the type hierarchy and is parent to the types such as `BeautyProduct`, `SportProduct` ...(Not shown in the figure). Each `Product` holds a list of `Discounts` they are applied. The OnlineShop has a user interface represented by the OnlineShopUI class, which is used to display information to the customers.

2.2 Unanticipated Extensions

A new feature is added to the online shop which requires creating an alert when a product is applied a surprise discount, also the list of discounted products should be available to the user at any time. The surprise discounts are submitted by Vendors and they can be submitted or withdrawn any time. In order to realize this extension in an OO-approach,

```
1  class ProductManager{
2    ...
3      Set<Product> surpriseDiscount = createSet();
4    public void submitDiscount(Product p, Discount d)
5    {
6      ...
7      if(d instanceof SurpriseDiscount)
8      {
9        surpriseDiscount.add(p);
10       OnlineShop.createDiscountAlert(p);
11     }
12   }
13   public boolean withdrawDiscount(Product p, Discount d)
14   {
15     ...
16     if(d instanceof SurpriseDiscount)
17       surpriseDiscount.remove(p);
18   }
19 }
20 class OnlineShop{//SINGLETON
21   ...
22   public void createDiscountAlert(Product p)
23   {
24     //create surprise discount alert for p
25   }
26   public void displayList(String listType)
27   {
28     if(listType.equals(''surprise'')
29       INSTANCE.getUI().display(ProductManager.surpriseDiscount);
30   }
31 }
```

Listing 1: A Java implementation of discount alert concern

we need to change several classes to host this extension. First the class ProductManager should keep a set of Products which are applied a surprise discount, in Listing 1 this is shown in line 3. This set is updated when a new discount of type SurpriseDiscount is submitted or withdrawn (lines 4– 18). There should also be some changes in the OnlineShop class. First a createDiscountAlert method should be added. Also the displayList method should be updated to include the surpriseDiscount list defined in the ProductManager class.

The OO-solution is scattered among ProductManager and OnlineShop classes and tangled with multiple methods. The code for the surprise discount concern and the book-keeping that comes with it creates cluttering. An aspect-oriented implementation can offer a better solution by encapsulating the concern in an aspect. In Listing 2 shows a possible solution. The set of products which are applied a surprise discount is kept in the aspect (line 1). The following two pointcuts submit and withdraw selects the products to which a SurpriseDiscount is applied (lines 2– 3). The corresponding advice declarations for these pointcuts maintain the surpriseDiscount set, also the submit join-point triggers the surprise discount alert method (line 7). There is also the display pointcut (line 4, which intercepts the call to displayList method and add the condition for the surprise discount list in an around advice (lines 13– 17). This aspect also includes an inter-type declaration which adds the createDiscountAlert method to the OnlineShop class.

Discussion and Requirements. AOP already helps to localize the concern and to add it without the need to modify

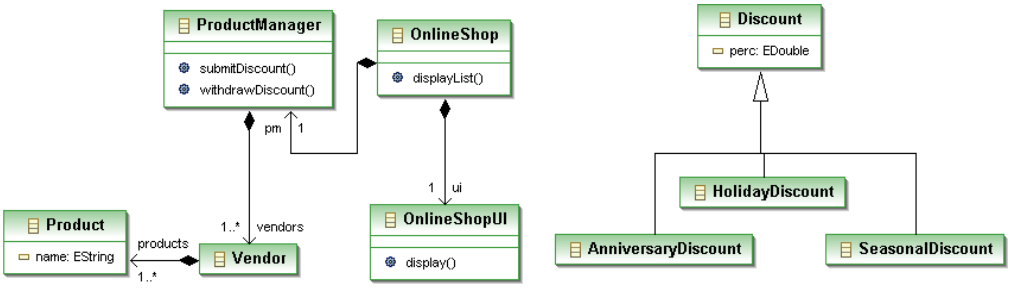


Figure 1: Part of an online shop application

```
1  Set<Item> surpriseDiscount = createSet();
2  pointcut submit(Product p): call(* ↙
       ProductManager.submitDiscount(..)) && args(p, ↙
       SurpriseDiscount);
3  pointcut withdraw(Product p): call(* ↙
       ProductManager.withdrawDiscount(..)) && args(p, ↙
       SurpriseDiscount);
4  pointcut display(String listType): call(* ↙
       OnlineShop.displayList(..)) && args(listType);
5  after(Product p): submit(p){
6    surpriseDiscount.add(p);
7    OnlineShop.instance().createDiscountAlert(p);
8  }
9  after(Product p):withdraw(p){
10   if(surpriseDiscount.contains(p))
11     supriseDiscount.remove(p);}
12 }
13 void around(String listType):display(listType){
14   if(listType.equals(''surprise''))
15     OnlineShop.instance().getUI().display(surpriseDiscount);
16   proceed(listType);
17 }
18 public void OnlineShop.createDiscountAlert(Product p){
19   //create surprise discount alert for a product
20 }
```

Listing 2: An Aspectj implementation of discount alert concern

existing code. However maintenance of the `surpriseDiscount` set requires the same boilerplate code the OO solution does (highlighted in gray). Essentially what the code does is select `Product` objects based on the discount they are applied to and deselect them once they are rid of this discount. This marks a phase in the life-cycle of a `Product` object, none of the presented solutions offer any declarative means to define such a life-cycle phase. Furthermore, reusing such existing sets by refining or composing them is not conveniently supported at all; e.g., if we want to find the subset of BeautyProducts of the surpriseDiscount set, 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 on 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 selecting/de-selecting objects according to the events they participate should be provided.

Requirement2: The selected objects should be maintained as a set, representing a *category* of objects.

Requirement3: The set of objects should be accessible and any changes to this set i.e. adding/removing objects should create a notification.

Requirement4: For the same kind of objects the sets should be composable to obtain new sets which also satisfy the preceding requirements.

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 contain AspectJ pointcut expression.

3.1 Instance Pointcut Language Features

A concrete instance pointcut definition consists of a left hand-side and a right-hand side (Figure 2, rule 1). 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*.

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 that are separated by the **UNTIL** keyword, defines the events and matching conditions (Figure 2, rule 1). 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

⟨*instance pointcut*⟩ ::= '`instance 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*⟩'`)`'

Figure 2: Grammar definition for instance pointcuts

remove which object: it selects the event that marks the end of the life-cycle phase of interest. When the remove expression is not defined, then the object is kept in the set until it *dies*.

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 are composed of expression elements. The expression elements can be one of two types, *before element* and an *after element* (Figure 2, rule 3-4). A sub-expression (add/remove expression) contains at least one *expression element* and at most two. In Figure 2 the second grammar rule depicts this statement.

In Figure 2 rules 3 and 4 contains the ⟨*pointcut−expression*⟩ rule which represents an AspectJ pointcut expression. However we have introduced a restriction that in every pointcut expression there must be a binding predicate (args, target etc.) that bind the **instance** parameter and the binding predicates are extended to include the **returning** clause. The **returning** clause binds the returned variable by a method or a constructor. AspectJ only allows it to be used with an after advice, specifically known as `after returning`.

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.

The instance pointcut in Listing 3 shows a basic example. The left-hand side of the instance pointcut indicates that the pointcut is called surpriseDiscount and it is interested in selecting Product objects. On the right hand side, there are two expressions separated by the **UNTIL** keyword. The first

```
1 static instance pointcut surpriseDiscount<Product>:
2   after(call(* ProductManager.submitDiscount(..))
3   && args(instance, SurpriseDiscount))
4   UNTIL
5   after(call(* ProductManager.withdrawDiscount(..))
6   && args(instance, SurpriseDiscount));
```

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

one is the add expression. It selects join-point marked by the method submitDiscount and from the context of this event it exposes the Product object with the **args** clause and binds it to the **instance** parameter. The second one is the remove expression and it selects the after event withdrawDiscount call and exposes the Product instance in the method arguments and binds it to the **instance**. This pointcut is the solution of the set maintenance problem presented in the motivation (section 2).

As well as the regular AspectJ binding predicates (**args, this, target**) we also support the usage of **`returning`** to bind the returned value of a method. If the **instance** parameter is bound by the **returning** clause then it has to be in an after event, since the returned value is only available after a method finishes execution.

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 formal definition of a multiset is as follows:

Definition 1. *A multiset is a pair (X, f), where X is a set and f is a function mapping X to the cardinal numbers greater than zero. X is called the underlying set of the multiset, and for any $x \in X$, $f(x)$ is the multiplicity of x.*

Note that we do not allow a multiplicity of zero or infinite.

The instance pointcut shown in Listing 4 selects Product instances, which are applied a `Discount`. The remove expression removes a Product instance if the `Discount` is removed from that `Product`. With this pointcut we would like to represent the currently discounted products. Multiset makes sure that Products can be added for each discount submission operation. When the same product is added with different types of discounts, and if one of discounts are removed, then still one entry of that instance is left in the set. If instance pointcuts only supported a set then as soon as a discount is removed from a product, its only copy would be removed and it would appear as if there are no more discounts on that product.

3.2 Refinement and Composition

```
static instance pointcut multi_discount<Product>:
  after(call(* ProductManager.submitDiscount(..))
  && args(instance))
  UNTIL
  after(call(* ProductManager.withdrawDiscount(..))
  && args(instance));
```

Listing 4: An instance pointcut utilizing multiset property

Instance pointcuts can be referenced by other instance pointcuts, they can be refined in terms of instance type and composed together to create new instance pointcuts.

3.2.1 Referencing and Type Refinement

Instance pointcuts are referenced by their names. 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 surpriseDiscount (Listing 3) can be refined as in the following:

```
static instance pointcut surpriseDiscountBeauty<BeautyProduct>:
  surpriseDiscount<BeautyProduct>
```

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

3.2.2 Instance Pointcut Expression Extension

In subsection 3.1.1 we have introduced the instance pointcut expression, which consists of a two sub-expressions (add and remove expression). The expression elements forming these sub-expressions can be accessed individually to be extended by concatenating other primitive pointcuts.

3.2.3 Instance Pointcut Composition

Instance pointcuts reify sets, for this reason the composition of instance pointcuts are defined in terms of set operations: *intersection and union.* In Figure 3, an extended version of the grammar definition is shown. A new instance pointcut can be defined by declaring an instance pointcut reference and a partial set composition expression. In order to declare a set intersection the keyword `inter` and to declare a set union the keyword `union` is used. Throughout the text we will use the mathematical symbols for these operations, ∩ as intersection and ∪ as union.

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

⟨*instance-pointcut*⟩ ::= 'instance pointcut' ⟨*name*⟩ '<'
⟨*instance-type*⟩ '>' ':' ...
| ⟨*ip-ref*⟩ ⟨*partial-set-composition*⟩*

⟨*partial-set-composition*⟩ ::= ⟨*set-operator*⟩ ⟨*ip-ref*⟩

⟨*set-operator*⟩ ::= 'inter'
| 'union'

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

Figure 3: Syntax for instance pointcut composition

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 the content of their sets *at the time of the composition.* The result is an instance pointcut which monitors the composing instance pointcuts and updates the composition sets accordingly. We have defined two composition operators for instance pointcuts, union and intersection. Since instance pointcuts are reified as multi-sets, these operations are different from the regular set operations.

Definition 2. *Assume* (X, f) *and* (Y, g) *are multisets.*

The ***intersection*** *of these sets is defined as* (V, h) *where,*

$$V = X \cap Y$$

and $\forall v \in V$ *the multiplicity of* v *is defined as*

$$h(v) = min(f(x), g(x))$$

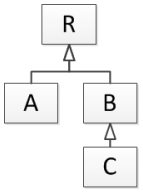
The ***union*** *of these sets is defined as* (Z, i) *where,*

$$Z = X \cup Y$$

and $\forall z \in Z$ *the multiplicity of* z *is defined as*

$$i(z) = max(f(x), g(x))$$

The objects in an instance pointcut's set can be added and removed at certain events, which makes the contents of the set dynamic. Therefore the composition operation requires monitoring of the component sets. Figure 5 presents a visualization of these operations. The small triangles represent the adding of an object to the instance pointcut set and the small circles represent the removal. The ellipses represent the instance pointcut set, and their major axes sit between



(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

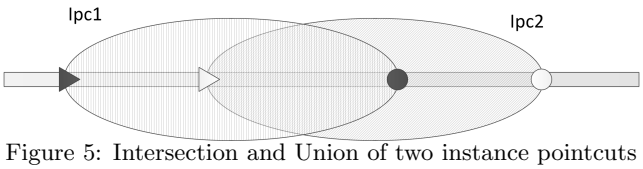


Figure 5: Intersection and Union of two instance pointcuts

the add and remove events of the corresponding instance pointcut (Ipc1 and Ipc2). The line represents execution flow of a *single* object. This object is first added to the set of Ipc1 at the first dark triangle. Between this event and Ipc2's add event (light triangle):

$$Ipc1 \cup Ipc2 = Ipc1$$

$$Ipc1 \cap Ipc2 = \varnothing$$

When the object is added to Ipc2's set at the light triangle then their intersection is no longer empty and the union no longer consists of only Ipc1's elements. The result of the union and the intersection is computed according to definition 2. When the object is removed from Ipc1's set (small dark circle) then the intersection is once again empty. Of course this abstract example only shows a simple case where an instance pointcut selects only one join-point for selecting and one joinpoint for removing an instance. In practice instance pointcuts can be more complex, where they can match multiple events which alters their sets.

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. 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 5 maintains a set of Products that are currently out of stock.

3.3.1 Set Access

Instance pointcuts reify a set and this set can be accessed through a static method, which has the same name as the instance pointcut identifier. Then the *read* methods of the collection 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  aspect MyAspect{
2    static instance pointcut outOfStock<Product>:
3      after(call(* Product.outOfStock(..)) &&
4            target(instance))
5      UNTIL
6      after(call(* Vendor.stock(..))
7            && args(instance));
8    ...
9  }
```

Listing 5: An instance pointcut for out of stock products

```
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 6: Calculate a damage estimate for out of stock products

In Listing 6, 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 method.

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 7, 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.

```
1 before(Product p): outOfStock_add(p)
2 {
3   OnlineShop.notifyVendor(p.getVendor, STOCK_MSG);
4 }
```

Listing 7: 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 by using JaMoPP[1] [3] as the foundation 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 refer to 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 8 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, respectively. 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 operations, both methods 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).

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 reasonable 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

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

Listing 8: Template of generated code for instance set management.

is bound by the **after returning** keyword in the advice definition instead of the pointcut definition. Therefore, the generated AspectJ code depends on which value is bound; this means when another pointcut is referenced then the binding predicate of the referenced pointcut should be known. Also it is not possible to work around this, using AspectJ's reflective **thisJoinPoint** keyword, as it 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 inline their definition.

For these reasons , our current implementation is based on the ALIA4J[2] [4, 5] implementation approach and framework for advanced dispatching languages. At its core, ALIA4J contains a meta-model of advanced dispatching declarations, called *LIAM*, 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*

[1] JaMoPP: Java Model Parser and Printer, see `http://jamopp.inf.tu-dresden.de`

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

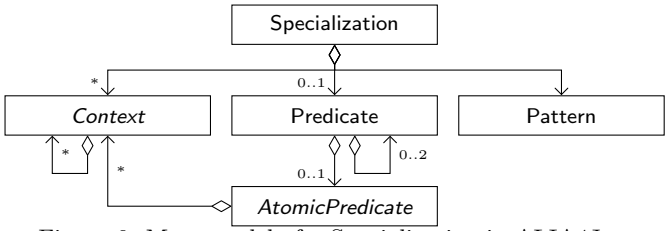


Figure 6: Meta-model of a Specialization in ALIA4J.

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

Listing 9: Example of an instance pointcut using composition and type refinement

```
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});
5
6      Set<Specialization> ipRefExpr = ipc1_add_before();
7      ipRefExpr = Util.addTypeConstraint(ipRefExpr, ${Type2});
8
9      ${ipc}_add_before = Util.orSpecializations(ipRefExpr, ↙
           simpleIPExpr);
10   }
11   return ${ipc}_add_before;
12 }
```

Listing 10: Generated code for creating the LIAM model for the add/before pointcut of the example instance pointcut.

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 [6]). 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 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.

For the example instance pointcut presented in listing 9, we show the code generated for the method creating the LIAM model for the add/before pointcut in listing 10; 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 9 and 10—the above mentioned library function.

What is generated from an instance pointcut reference is shown in line 6: the LIAM model of the referenced instance pointcut is invoked. 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.

The disjunction performed in line 9 of listing 10 corresponds to the usage of the operator in line 2, listing 9. 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 11 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.

To ensure this consistency also between AspectJ advice and our implementation of instance pointcuts, the AspectJ point-

```
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 11: Deployment of the bookkeeping for an instance pointcut.

cut-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. EVALUATION

5. RELATED WORK

AO-extensions for improving aspect-object relationships are proposed in several studies. Sakurai et al. [7] 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 [8] 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. [9] 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 [10] 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 [11]. 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.

6. 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, which allows reuse of existing instance pointcuts and consistency among corresponding multisets. Finally, we have presented our compilation approach for instance pointcuts based on the generation of LIAM models and 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 [12] or Composition Filters [13], the point-in-time join point model [2]) would influence the notation of these expressions.

ALIA4J provided us with the flexibility AspectJ lacked in instance pointcut composition and type refinement. By transforming instance pointcuts to LIAM models, we were able to reuse features offered by ALIA4J and extend some parts of the framework according to our needs. This allowed us to implement a uniform code generator which outputs composable code.

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 [14]. 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.

7. 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] Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the gap between modelling and java. In van den Brand, M., Gaševic, D., Gray, J., eds.: Software Language Engineering. Volume 5969 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 374–383 10.1007/978-3-642-12107-4_25.

[4] Bockisch, C., Sewe, A., Yin, H., Mezini, M., Aksit, M.: An in-depth look at alia4j. Journal of Object Technology **11**(1) (2012) 1–28

[5] Bockisch, C., Malakuti, S., Aksit, M., Katz, S.: Making aspects natural: events and composition. In: Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD, ACM (2011) 285–300

[6] Bockisch, C., Mezini, M.: A flexible architecture for pointcut-advice language implementations. In: Proceedings of the 1st Workshop on Virtual Machines and Intermediate Languages for Emerging Modularization Mechanisms, VMIL, ACM (2007)

[7] 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

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

[9] 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

[10] 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)

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

[12] 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

[13] 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)

[14] 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