SLE2012

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

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. Often 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. **requirement: reify set of objects** 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 Example Section - Currently Nameless

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` and a `WishList` which may contain arbitrary products. There are two customer types in the system, `RegularCustomer` and a `GoldCustomer`. `GoldCustomers` can benefit from additional discounts when buying certain products. 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. A customer can add/remove items from his shopping bag. If a product is added to the bag, any discounts associated with that product is applied by the `ProductManager`. 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.

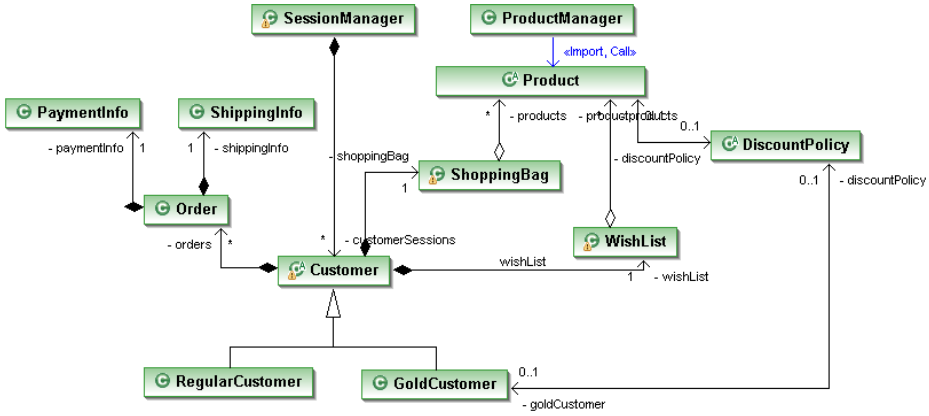


Fig. 1. A simple online shop application

Let us take a closer look at the type `Product` and the kind of events its instances can be involved in. A `Product` instance can be added/removed from a shopping bag. When it is added to a shopping bag its price may be decreased by applying a discount policy. When a new discount policy is associated with a product, the customers who have that products in their wish-lists can be notified. These examples contain an initial event that is associated with a particular instance, which triggers a follow-up event.

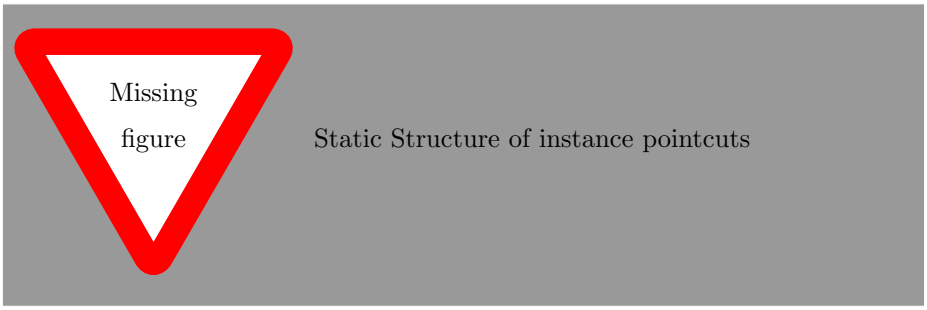
Explain what cannot be done with current approaches

Explain briefly How instance pointcuts can remedy the situation

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 Features

3.2 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.

3.3 Supported AspectJ Predicates

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

3.4 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

3.5 Deselection

How to deselect instances, what are possible checks?

Example using UNTIL clause, also incorporating before/after

3.6 Instance Pointcut References

how to reference instance pointcuts

Type-refined references

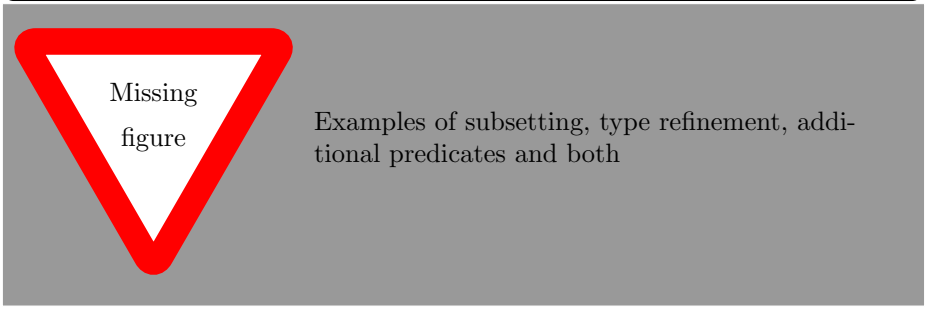
3.7 Instance Pointcut Composition

3.8 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



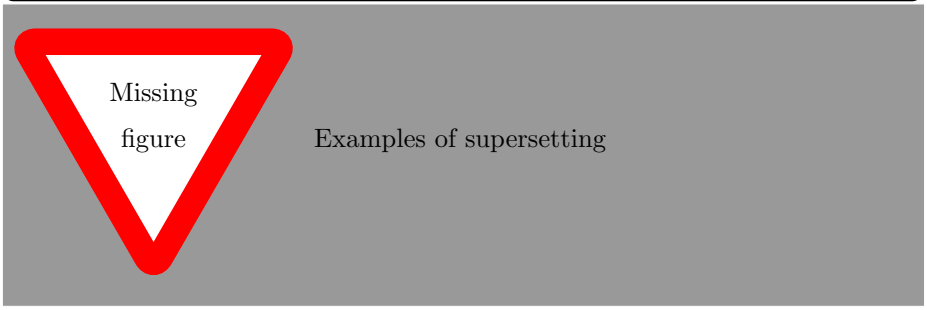
Examples of subsetting, type refinement, additional predicates and both

3.9 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

3.10 Union, Intersection and Set Difference

Define these operations and give simple examples

3.11 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 `shapes` can be refined with the following syntax: `shapes<Rectangle>`. This indicates we would like to

select the subset of `Rectangle` instances from the set of `Shape` instances selected by `shapes`. Note that this notation will also select subtypes of `Rectangle`. If this is not the desired effect then the following: `instance pointcut rectangle< Rectangle>: shapes && if(instance.getClass().equals(Rectangle.class))` can be used. The effect of a refinement subset being empty is equivalent to that of an unmatched pointcut. Note that if the + operator was not used in the pointcut expression, the refinement expression would still be legal, since `Rectangle` is a subtype of `Shape`. However it would result in a warning explaining the `Shape`'s subtypes are not selected.

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 `shapes` pointcut, we can define the following:

```
1 instance pointcut rectangle<Rectangle>: shapes<Rectangle> &&
      if(instance.getWidth() > 10)
```

The `rectangle` instance pointcut is interested in `Rectangle` objects which are selected by `shapes` and which has a width larger than 10. The pointcut expression to select these instances is very concise and is shown in the listing above. `shapes<Rectangle>` statement selects the `Rectangle` instances selected by `shapes` pointcut, then the `if` pointcuts checks whether these instances satisfy the condition. Note that the following statement; `instance pointcut circle< Circle>: shapes<Rectangle> && if(instance.getWidth()> 10)` , would result in a compile error for two reasons. First we cannot assign `Rectangle` instances to a `Circle` instance pointcut and secondly `instance.getWidth()` is illegal since `instance` is the implicit variable of the `circle` pointcut and it has the type `Circle`.

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.

3.12 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.

```
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?

3.13 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 3.13 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 3.12). 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`.

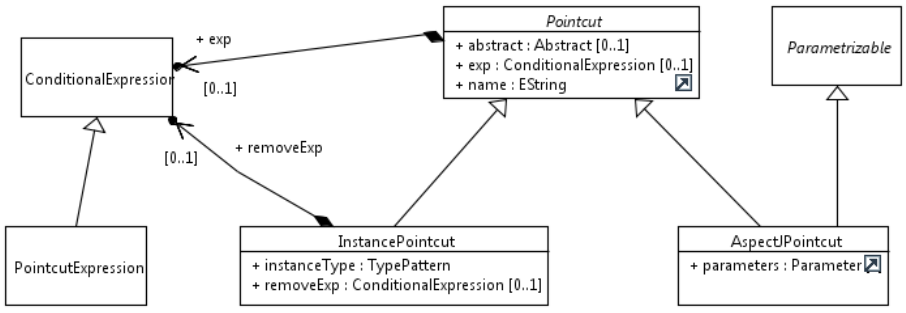


Fig. 2. Instance pointcut static structure

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 1.1.

Listing 1.1. Code generation templates for instance pointcut to AspectJ generation

```
Enumeration ExpressionType {SELECT, REMOVE}
generate(InstancePointut pc){
  if(pc.isAbstact){
    'abstract pointcut' pc.name'('pc.instanceType 'instance)'
    'Set<'pc.instanceType'>' pc.name'_set'
  }
  else{
    'Set<'pc.instanceType'>' pc.name'_set = new TreeSet<'pc.
       instanceType'>();'
    generateAspectJCode(pc, SELECT);
    if(pc.removeExpression != null)
      generateAspectJCode(pc, REMOVE);
  }
}
generateAspectJCode(InstancePointcut pc, ExpressionType eType
   ){
  String pcname, setOperation, setName = pc.name + '_set';
  PointcutExpression expTemp;
  switch(eType){
    case REMOVE:{
      pcname = pc.name + '_remove';
      setOperation = 'remove';
      expTemp = pc.removeExp;
    }
    case SELECT:{
      pcname = pc.name;
      setOperation = 'add';
      expTemp = pc.exp;
    }
  }
  if(exp.contains(ReturningStatement))  {
     PointcutExpression newExp = expTemp.remove(
        ReturningStatement);
    'pointcut' pcname'():' print(newExp) ';'
    'after() returning(' instanceType 'instance):' pcname'(){
       '
  }
  else{
    'pointcut' pcname'('pc.instanceType 'instance):' print(
       expTemp) ';'
    'after(' pc.instanceType 'instance):' pcname'(instance){'
  }
  'boolean flag ='setName.setOperation '(instance);
   if(flag)
    print(instance  +' setOperation ');
  }'
}
```

4 Application on Online Shop

this section will include sequence diagrams and advanced instance pointcuts for showing their benefits....

5 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?

6 Related Work

7 Discussion

8 Conclusion and Future Work

keep it short