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

Components interact through their interfaces. In component-based systems, there are cases where a certain functionality is provided by a third-party software which usually comes with an incompatible interface, which makes connecting new components and the existing components a challenge. An important requirement of *binding* two components is to encapsulate the binding declarations in a separate module, while leaving the bound components untouched. In this study we present a language which satisfies this requirement and offers a reusable, maintainable and concise way of expressing binding.

Our language is composed of two main structures; **instance pointcuts** and **adapter declarations**. Instance pointcuts are specialized pointcuts, which are used to capture the instances of a type, which at some point in their life cycle become relevant. This can be creation of the instance, calling a certain method on the instance or passing the instance as an argument. The conditions for becoming relevant is defined in the pointcut expression. Adapter declarations provide a declarative syntax for implementing object adapters, where the objects to be adapted are selected by instance pointcuts. The benefit of our approach partly comes from utilization of Aspect Orientation (AO), which allows modularization of binding concern. But the major benefit is due to the marriage of two new language concepts which take object adapters to a new level.

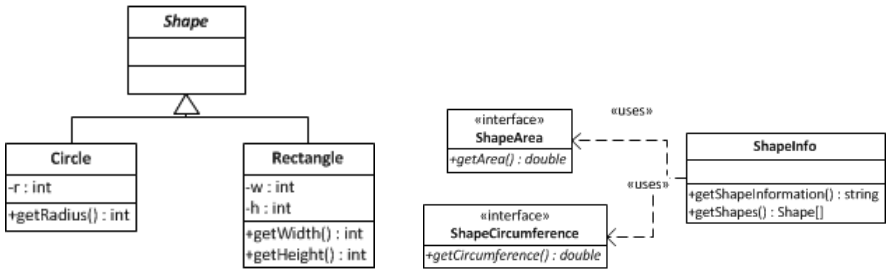
The traditional object adapter is shown in Figure *(objectadapter)*. Instance pointcuts provide means to select a subset of instances that belong to a specific type. It is also possible to select all subtype instances of a supertype, since the instance pointcut captures dynamic types. In adapter declarations this subset can be adapted by referring to the instance pointcut. An adapter declaration consists of a unique name, an instance pointcut reference and a list of interfaces to be implemented. The interface methods are overriden in the adapter declaration, referring to its *adaptee* where adaptee is an instance belonging to the set of instances selected by the instance pointcut. When an instance pointcut expression matches a join-point and that instance pointcut is referred to by an adapter declaration, then an adapter instance is automatically created containing the instance in the matched join-point. In an OO approach this would require adapter instantiations at various points in the code, which will cause tangling of adapter instantiation concern.

It is also possible to have an inheritance hierarchy among adapters, by defining *abstract adapters*. Abstract adapters do not have to implement all the interfaces they declare, whereas *concrete adapters* have to provide an implementation for every interface they declare or for the unimplemented interface declared by their super-adapter. Concrete adapters can also override the implementations of their super-adapters. This abstraction mechanisms leads to maintainable adapters and reduce programming efforts should the components evolve.

The AO nature of our approach allows all of these features to be encapsulated in an aspect. The concern is localized therefore there's no scattering and tangling. Moreover a run-time library for querying and retrieving adapters is created. With this library the user can query adapters by using an instance, a type, an adapter name or a combination of these as a key. This allows accessing adapters in a flexible way, whereas in the traditional OO implementation, it is only possible to access an adapter is by using its instance name.

2 The Binding Language

Instance pointcuts and adapter declarations are language concepts and ideally every aspect-oriented language can be extended to host them. We have implemented a prototype in AspectJ. While we have extended the syntax of AspectJ for instance pointcuts and adapter declarations, we have reused AspectJ's joinpoint model in instance pointcut expressions. We will discuss possible extensions in other well known AO-languages at the end of this section.



(a) Shapes Hierarchy (b) ShapeInfo and used interfaces

Fig. 1. Shapes Example

We will explain the basics of the language with the help of a simple example. In Figure 1(a) a simple hierarchy of shapes is shown. The abstract class `Shape` has two subtypes called `Circle` and `Rectangle`. The interfaces provided by these classes can be seen in the figure. The `ShapeInfo` class shown in Figure 1(b) collects the area and the circumference information of a given `Shape` using `ShapeArea` and `ShapeCircumference` interfaces, which are not implemented by any of the classes in the `Shape` hierarchy.

Listing 1.1. An instance pointcut selecting Circle objects and an adapter declaration using this pointcut

```
1  aspect ShapeAdapterAspect{
2    instance pointcut circles<Circle> : call(* Circle.new(..))
         && returning(instance);
3
4    declare adapter: CircleAdapter{ShapeArea,
         ShapeCircumference} adapts circles{
5      public double getArea()
6      {
7        return Math.PI*adaptee.getRadius()*adaptee.getRadius();
8      }
9
10     public double getCircumference()
11     {
12       return 2*Math.PI*adaptee.getRadius();
13     }
14   }
15 }
```

In Listing 1.1 an simple instance pointcut that selects `Circle` instances and an adapter using that pointcut is defined. The name of the instance pointcut (line 2) is **`circles`** and the instance type is shown in Java generics syntax as **`<Circle>`**. The `call` primitive pointcut selects the join points where a new `Circle` object is created,then **`returning(instance)`** binds the returned `Circle` instance to the *implicit variable* **`instance`**. The complete set of features of instance pointcuts will be discussed in Section 2.1. The adapter declaration (line 4), defines an adapter called `CircleAdapter` which implements `ShapeArea` and `ShapeCircumference` interfaces, indicated inside the curly braces. After the **`adapts`** keyword we declare which set of object we would like to adapt, in this case the set of objects selected by the `circles` pointcut. In the body of an adapter declaration, the implementation of the declared interfaces is given. Notice the **`adaptee`** keyword in the method bodies (lines 7, 12). The `adaptee` keyword refers to the object that's being adapted, in this case a `Circle` object, therefore can access its methods.

2.1 Instance Pointcuts

Instance pointcut is a declarative language construct that is used to select a group of objects of a specific type. The result of the selection is a *set*, hence it cannot contain the same object more than once. Although throughout the paper we use instance pointcuts with adapters, they can be used on their own and they offer further features for managing object sets.

An object can be accessed from multiple modules during its life-cycle. 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. For example, in an online-store application, during a purchase action a *product* object is used by modules like product manager, shopping bag, shipping info and payment process. For analysis purposes, we would like to count the purchases that are made from a particular city. Later according to our analysis results, we may want to apply a discount to that product type and apply further discount if 3 items of that product is present in the shopping bag. To perform these operations we need to insert code to multiple modules to capture relevant events, which cross-cuts the original module functionality.

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. For the online store example, instance pointcuts can select necessary object for the relevant criteria such as shipping destination, price etc. in a single concise pointcut declaration.

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 shapes<Shape>: call(* Shape+.new(..)) &&
    within(ShapeGenerator1) && returning(instance)
```

The instance pointcut above shows a basic example. The left-hand side of the instance pointcut indicates the pointcut is called `shapes` and it is interested in selecting the `Shape` objects. On the right-hand side, the pointcut expression selects the join-points where `Shape` or its subtypes are instantiated (+ operator indicates subtypes are selected, for a complete documentation please refer to ...) *within* the class `ShapeGenerator1`. The created instances are exposed and bound by the `returning` clause to *implicit variable* `instance`, which represents a member of `shapes`'s set. After selecting a set of objects, instance pointcuts offer ways to manipulate this set.

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.

What is the benefit of refinement?

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.

```
1 instance pointcut parallel<Parallelogram>: call(*
    Parallelogram.new(..) && returning(instance) UNTIL call(*
     Parallelogram.setWidth(..)) && if(instance.getWidth() >
    10)
```

Listing 1.2. An Adapter Hierarchy

```
1  aspect ShapeAdapterAspect{
2    instance pointcut shapes<Shape> : call(* Shape+.new(..)) &&
         returning(instance);
3
4    declare adapter: abstract ShapeAdapter{ShapeArea,
         ShapeCircumference, ShapeArea} adapts shapes{
5      public String getColor(){
6        if(this.getArea() < 40)
7          return ``RED'';
8        else
9          return ``BLUE'';
10     }
11   }
12
13   declare adapter: CircleAdapter extends ShapeAdapter adapts
         shapes<Circle>{
14     //Implementation of ShapeArea and ShapeCircumference
15   }
16   declare adapter: RectangleAdapter extends ShapeAdapter
         adapts shapes<Rectangle>{
17     //Implementation of ShapeArea and ShapeCircumference
18   }
19 }
```

2.2 Adapters

2.3 Adapter Hierarchies

In this section we will add a new required interface to the example presented in Figure 2, called `ShapeColor`. This interface has a `getColor()` method, which returns `RED` if the area of the shape is smaller than `40` and `BLUE` for the rest.

In Listing 1.2 an adapter hierarchy is shown. We have modified the instance pointcut to capture all `Shape` instances including instances of its subtypes using the "+" operator of AspectJ(line 2). The implementation of the `ShapeColor` interface does not depend on the instance type. We provide the implementation of this interface in an **abstract adapter** called `ShapeAdapter`(line 4) , which adapts all the `Shape` objects that are created. An adapter is declared abstract by placing the `abstract` modifier before its name. *An abstract adapter can have unimplemented methods, whereas a concrete adapter had to implement all the unimplemented interface methods it declares and inherits.* Since `ShapeAdapter` is abstract it does not have to implement `ShapeArea` and `ShapeCircumference`, however it can refer to these interfaces' methods, as seen in line 6. Abstract adapters cannot be instantiated, therefore it is necessary to have sub-adapters that are concrete.

The concrete adapters extending `ShapeAdapter` are `CircleAdapter` (line 13) and `RectangleAdapter` (line 16). There are two things to notice in these sub-adapter declarations. First is that they do not declare any interfaces, that information is inherited from `ShapeAdapter` declaration. The second is the way instance pointcut `shapes` is refined. *Instance pointcuts store the dynamic types of instances.* The statement `shapes<Circle>` selects a subset of `Shape` objects which have the dynamic type `Circle`. This refinement mechanism lets us create a `CircleAdapter` without writing a new pointcut which explicitly selects `Circle` instances, also by refining the same pointcut used in the super-adapter we ensure the set of `Circle` objects are a subset of `Shape` objects selected by `shapes` instance pointcut.

I think we should discuss whether we allow using a different pointcut in the sub-adapters.

2.4 Adapter Instantiation and Retrieval

While adapters are widely used to resolve incompatible interface problem, they introduce a cross-cutting concern to the which we identify as *adapter instantiation concern.* In an OO-approach the `ShapeInfo`(Figure 1(b)) class has to create or reuse a `CircleAdapter` whenever it wants to get the area of a `Circle` object. In such a small example this does not cause a hindrance. However if we increase the size of this problem into a thousand `Shape` instances and increase the number of interfaces ShapeInfo has to deal with to 20, then instantiation of the right adapters for the right shapes and managing these adapters is not so trivial anymore.

Instance pointcuts and adapters declarations provide a declarative way for matching instances with the appropriate adapters. When an instance pointcut matches an instance and there's a concrete adapter declaration for that instance pointcut, the adapter instance containing the matching instance is automatically created. The automatic creation of adapters is an implicit behavior and it modularizes the adapter instantiation concern.

3 Related Work