Chapter 1

Instance Pointcuts

1.1 Using Instance Pointcuts with Design Patterns

Instance pointcuts offer a new way of modularizing objects; this new modularization method can be used in conjuction with design patterns for a well-localized implementation and fine-grained assignment of object roles. In this section we will discuss how instance pointcuts improve the implementation of design patterns. We will go over three scenarios which use factory, adapter and observer patterns respectively.

Let us first introduce the example setting for the different scenarios. We have a simple drawing application which can be used to draw basic shapes. The user selects a certain shape, which appears in the middle of the screen. Then the user can scale, move and color the shape as she pleases. The shape objects adhere to the hierarchy shown in figure 1.6.

The `Shape` class is an abstract class, and root of the shapes hierarchy. The hierarchy contains two types of shapes, `Circle` and `Polygon`. The `Polygon` class have common shapes as concrete subclasses, `Triangle` and `Rectangle`, the latter has a subclass called `Square` which implements the `RegularPolygon` interface. Each concrete shape object is created by a factory class that implements the abstract factory `ShapeFactory`. The `ShapeFactory` interface contains a single method called `createShape` which returns a `Shape` object. According to the selection made in the application GUI, the corresponding shape factory's `createShape` method is invoked. For example if the user selects the shape circle, then the `CircleFactory`'s creation method is called and the returned object is drawn on the screen.

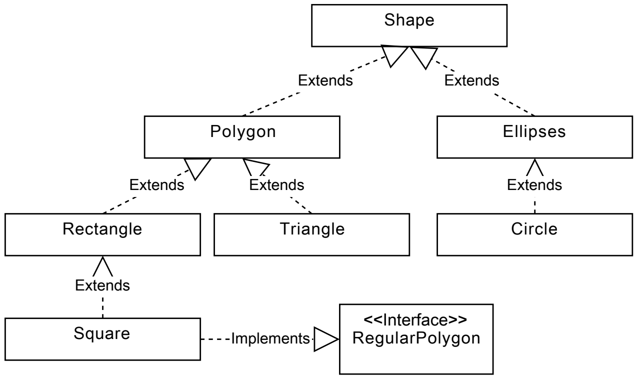

Figure 1.1 – The shapes hierarchy

```
int count = 0;
if(selected.isCircle())
 for(Shape s: shapes)
  if(s instanceof Circle)
   count++
```

Listing 1.1 – A code snippet for counting `Circle` objects

1.1.1 Scenario 1 – Factory Pattern

In this scenario we would like to extend our application with the following feature; when the user selects a certain type shape and presses the button “count”, the application will display how many shapes of that type is currently in the canvas.

With the current implementation, each shape that is drawn is kept in a list called `shapes<Shape>`. In order to count, for example, `Circle` objects, we need to add the code shown in listing 1.22. This code counts the `Circle` objects in the list `shapes`, every time the a circle is selected and the count button is pressed. In order to count for different shapes, we need to add additional if statement to identify the type of the selected shape.

Rather than counting everytime the count button is pressed, we can also keep track of the number of the objects in designated variables. For example, we can have an static integer field called `circleCount`. Whenever the `CircleFactory` creates a new `Circle` it can increment this value with the following statement `Canvas.circleCount++`. In the event of deleting a `Circle` object, the method `delete()` is called on the object. So inside the delete method of the `Circle` class we must call `Canvas.circleCount--`. This is obviously a scattered implementation of the counting feature.

```
1 static instance pointcut circles<Circle>:
2  after(* CircleFactory.createShape(..) &&
       returning(instance))
3  UNTIL
4  after(* Circle.delete() && target(instance));
```
Listing 1.2 – An instance pointcut that keeps track of `Circle` objects

```
int count = 0;
if(selected.isCircle())
  for(Shape s: shapes)
    if(s instanceof Circle)
      {
   if(s.getFilling() instanceof Texture)
    count++
      }
```
Listing 1.3 – Extended code snippet for counting textured or colored `Circle` objects

With instance pointcuts we can localize the counting operation by creating an instance pointcut for each shape. An example is shown in listing **??**. The instance pointcut `circles` is responsible for keeping a set of `Circle` objects, while the life-cycle phase for each object is marked by its creation (line 2) to its deletion (line 4). Using this instance pointcut, the number of `Circle` objects can be obtained by `[Aspect].circles().size()`.

Modularizing the shape groups concern in instance pointcuts is also beneficial for possible future extensions. For example we want to extend our application with a selection feature as follows; when the user selects a single shape object and double clicks on it, then the other objects which are of the same type must be selected. If the user performs this operation on a circle shape, we can retrieve the set of objects maintained by `circles` instance pointcut and iterate over it to set the field `selected` to `true`.

Another case when the information about the origin of the object creation bears a significance is when there are multiple ways of creating an object; for example when there are multiple factory classes or methods which create a differently configured version of objects of same type.

For example, assume we have replaced the `CircleFactory` class with two new factory classes called `TexturedCircleFactory` and `ColoredCircleFactory`, which create `Circle` objects with different values for the `filling` field. Going back to the shape counting example discussed above, let us consider once again listing 1.22. Here the `instanceof` check on line 4 cannot distinguish between a `Circle` object that is textured or colored. So if we would like to count textured circles, we need to extend this code as shown in listing **??**. On line 6 we perform an additional check to see if the `filling` of the `Circle` object is an instance of the type `Texture`. An instance pointcuts solution would be a slight modification of the instance pointcut shown in listing 1.21; instead of selecting the join-points at `CircleFactory.createShape`, we select `TexturedCircleFactory.createShape`.

In this scenario we have shown that instance pointcuts can be used in conjunction with the factory pattern to capture the information "by which factory an object was created". We have discussed two benefits; the localization of concerns by using the modularization offered by instance pointcuts and capturing information about the configuration of the object by looking at its origin of creation.

1.1.2 Scenario 2 – Adapter Pattern

In this scenario, we want to add the export feature to out drawing application to a different file format. The new file format can only support non-textured shapes with four edges or more, so even if we have drawn circles in a drawing, we cannot save them as circles. It is also not possible to save the triangles, because of the four edges limitation. If the user chooses to do so, she can save them as squares which have the same edge size as the diameter of the circle.

In order to implement this feature we need to single out non-textured rectangles (including squares) and we need to adapt the non-textured `Circle` objects to `Square` objects, to save them. Listing 1.23 shows a simple `Circle` to `Square` object adapter.

Since we can only save a specific set of objects, i.e non-textured polygons which have more than four edges, we need to filter these object from the canvas that is being exported. The first instance pointcut shown in listing 1.24 keeps a set of non-textured objects. The following two instance pointucts refines this set according to type; `nonTexturedCircles` selects non-textured `Circle` objects and `nonTexturedRect` keeps non-textured `Rectangle` objects.

Note that the non-textured `Circle` objects should be adapted before the export operation. By using the monitoring features of instance pointcuts, we can create the adapted values inside the same aspect that contains the instance pointcuts mentioned above. In listing 1.25 an implementation that uses `Circle2SquareAdapter` is shown. In this implementation we create a `Map` called `c2sMap` (line 1) that will hold the `Circle` objects on the canvas and their corresponding `Square`

```
public class Circle2SquareAdapter extends Square
{
  Circle adaptee;
  public Circle2SquareAdapter(Circle adaptee)
  {
    this.adaptee = adaptee;
  }
  @Override
  public double getEdgeLength()
  {
    return adaptee.getDiameter()
  }
}
```

Listing 1.4 – An adapter for creating a `Square` object by wrapping a `Cicle` object

```
static instance pointcut nontexturedshapes<Shape>:
  after(* Shape.new(..) && returning(instance) && !within(*
      Textured*Factory))
  UNTIL
  after(* Shape.delete() && target(instance));
static instance pointcut nonTexturedCircles<Circle> :
    nontexturedshapes<Circle>;
static instance pointcut nonTexturedRect<Rectangle> :
    nontexturedshapes<Rectangle>;
```

Listing 1.5 – Instance pointcuts that select non-textured shapes in the canvas

```
static Map<Circle, Square> c2sMap = new
    HashMap<CircleSquare>();

after(Circle c): nonTexturedCircles_instanceAdded(c);
{
  c2sMap.put(c, ((Square)new Circle2SquareAdapter(c)));
}
after(Circle c): nonTexturedCircles_instanceRemoved(c)
{
  c2sMap.remove(c);
}
public static Collection<Square> getAdaptedCircles()
{
  return c2sMap.values();
}
```

Listing 1.6 – Adapting non-textured `Circle` objects to `Square` objects

```
public void export()
{
  List<Rectangle> 2bExported = new ArrayList<Rectangle>();
  2bExported.addAll([Aspect].nonTexturedRect());
  2bExported.addAll([Aspect].getAdaptedCircles());
  Exporter.export(2bExported);
}
```

Listing 1.7 – The export function

represenation using the `Circle2SquareAdapter`. In order to populate this map we use the `nonTexturedCircles_instanceAdded` pointcut, which is readily available with the definition of `nonTexturedCircles`. In the after advice that uses this pointcut (lines 3– 6) performs the following operation; when a new `Circle` object is added to the canvas, a new `Square` object is created by adapting this `Circle` object. Then these objects are added to the map, `Circle` object as the key and `Square` object as the value. The `nonTexturedCircles_instanceRemoved` pointcut is used to remove the mappings from `c2sMap` (lines 7– 10). Finally we create the static method `getAdaptedCircles` to return the adapted values currently contained in the map `c2sMap`. Using these instance pointcuts and the extra code shown in listing 1.25, we can implement the export function as shown in listing 1.26.

In this scenario we have selected a specific set of objects and adapted them with a given object adapter. Instance pointcuts provided us with the expressive power over selecting specific objects as adaptees. Also we localized the adaptation concern to the aspect, which prevented a possible tangling in the export function.

1.1.3 Scenario 3 – Observer Pattern

Our final scenario shows how instance pointcuts can help localize the assignment of the role of *being observed* to an object. In this scenario we want to observe the color change events of `Triangle` objects. Using instance pointcuts we can select the set of colored `Triangle` objects as shown in listing 1.27 (lines 1– 4). We monitor the addition of colored triangles to the canvas with the `triColor_instanceAdded` pointcut (line 6). To each colored triangle, we add the `ColorChangeObserver` after it is added to the canvas (line 8).

Once again instance pointcuts provide expressive power over the choice of objects that is observed. Also it localizes the code for adding an observer to the observer list of an object, in an aspect. The advantage is a fine-grained management of objects in the application

```
static instance pointcut triColor<Triangle>:
  after(* ColoredTriangleFactory.createShape(..) &&
      returning(instance))
  UNTIL
  after(* Triangle.delete() && target(instance));

after(Triangle t): triColor_instanceAdded(t)
{
  t.addObserver(new ColorChangeObserver());
}
```

Listing 1.8 – An instance pointcut that keeps track of colored `Triangle` objects

and the prevention of implementation related issues that comes with using the observer design pattern.

Bibliography

[1] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes Virtual Machine Research Project: Building an Open-Source Research Community. *IBM Systems Journal*, 44, 2005.

[2] L. M. Bergmans and M. Aksit. How to deal with encapsulation in aspect-orientation. In *Proceedings of OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[3] C. Bockisch and M. Mezini. 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.

[4] C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at ALIA4J. *Journal of Object Technology*, 11(1):1–28, 2012.

[5] E. Bodden. Stateful breakpoints: a practical approach to defining parameterized runtime monitors. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 492–495, New York, NY, USA, 2011. ACM.

[6] E. Bodden, R. Shaikh, and L. Hendren. Relational aspects as tracematches. In *Proceedings of the 7th international conference on Aspect-oriented software development*, pages 84–95. ACM, 2008.

[7] R. Chern and K. De Volder. Debugging with control-flow breakpoints. In *Proceedings of the 6th international conference on Aspect-oriented software development*, AOSD '07, pages 96–106, New York, NY, USA, 2007. ACM.

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

[9] R. DeLine and M. Fähndrich. Typestates for objects. In M. Odersky, editor, *ECOOP 2004 - Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 465–490. Springer Berlin Heidelberg, 2004.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.

[11] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. Escala: modular event-driven object interactions in scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 227–240, New York, NY, USA, March 2011. ACM.

[12] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

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

[14] K. Kawauchi and H. Masuhara. Dataflow Pointcut for Integrity Concerns. In B. de Win, V. Shah, W. Joosen, and R. Bodkin, editors, *AOSDSEC: AOSD Technology for Application-Level Security*, Mar. 2004.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. *ECOOP 2001 Object-Oriented Programming*, pages 327–354, 2001.

[16] H. Masuhara, Y. Endoh, and A. Yonezawa. A fine-grained join point model for more reusable aspects. In N. Kobayashi, editor,

Programming Languages and Systems, volume 4279 of *Lecture Notes in Computer Science*, pages 131–147. Springer Berlin / Heidelberg, 2006. 10.1007/11924661_8.

[17] D. Pearce and J. Noble. Relationship aspects. In *Proceedings of the 5th international conference on Aspect-oriented software development*, pages 75–86. ACM, 2006.

[18] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matuura, and S. Komiya. Design and implementation of an aspect instantiation mechanism. *Transactions on aspect-oriented software development I*, pages 259–292, 2006.

[19] H. Yin, C. Bockisch, and M. Aksit. A pointcut language for setting advanced breakpoints. In *Proceedings of Aspect-Oriented Software Development*. ACM, 2013.