Grouping Objects using Aspect-Oriented Adapters

Stefan Hanenberg, Rainer Unland

Institute for Computer Science
University of Essen, D - 45117 Essen
`{shanenbe, unlandR}@cs.uni-essen.de`

Abstract. Aspect-Oriented Programming (AOP) is an approach for realizing separation of concerns and allows different concerns to be weaved into existing applications. Concerns usually cross-cut the object-oriented structure. Whenever a concern needs to invoke some operations on objects of the given structure the problem arises, that those objects have different types, but the concern expects them to be handled in the same way. Therefore a mechanism for grouping objects of different types is needed.This paper discusses different mechanisms and proposes aspect-oriented adapters for grouping types and shows how this approach permits a higher level of flexibility and reduces the limitations of known approaches. Aspect-oriented adapters are not limited to a specific general purpose aspect language (GPAL). Nevertheless the examples in this paper are realized in AspectJ, which is by far the most popular and well-established general purpose aspect language.

1 Motivation and Problem Description

Let us assume we want to make objects persistent, which are created by an existing simulation-application. As pointed out in [3] persistency is a concern and so this is a typical application of Aspect-Oriented Programming [4]. Every newly created object should be added to a persistent storage and whenever the state of a certain object changes, its representation on the store must be updated. There is no need to offer an interface for retrieving objects, because the simulation itself does not use former objects. Instead the information is used by another application which directly accesses the storage for retrieving information about the simulation. The objects to be stored are all instances of class `Point`.

A suitable (straight-forward) solution for this problem in AspectJ [5] would be an aspect, which writes the state to the store every time an object is created and whenever its state changes (fig. 1). An instance of the aspect `PersistentPoint` is created for every `Point` instance. The aspect generates an object id (realized as an instance counter) and stores it in its attribute `id`. After creating a new `Point` the object is written to the persistent storage realized in the constructor of `PersistentPoint`.

The state of a point changes, whenever the methods `setX()` or `setY()` are invoked. Therefore a *pointcut* `setPC()` is defined for any instance of `Point` receiving a set-message. Whenever this happens the corresponding *pointcut method* (or *advice* in the AspectJ terminology) is executed which reads a point's state (`getX()`, `getY()`) and updates the persistent storage.

```
class Point {
  private float x=0;
  private float y=0;
  public float getX() {
    return x;}
  public float getY() {
    return y;}
  public void setX(float x) {
    this.x = x;}
  public void setY(float y) {
    this.y = y;}
}
```

```
aspect PersistentPoint
      of eachobject(instanceof(Point)){
  private static int idnum = 0;
  private int id = ++idnum;
  public PersistentPoint() {
    .. write new (unitialized) object to storage}
  pointcut setPC(Point p): instanceof(p) &&
           (receptions(void setX(float)) ||
            receptions(void setY(float)));
  after(Point p):  setPC(p) {
    float x = p.getX());
    float y = p.getY());
    ...update x,y of object id}
}
```

Figure 1: a) Class `Point`, b) Aspect `PersistentPoint`

Let us assume there is another (similar) application having its own implementation of a point `AnotherPoint` identical to `Point`. The proposed solution directly depends on the class `Point` and cannot be used for other classes. Therefore it would be more desirable to define a persistency aspect without being limited to class `Point`.

AspectJ supports inheritance relationships between aspects and allows to declare abstract aspects, so it seems to be a good choice to define an abstract aspect `PersistentObject`, which is responsible for creating the object id and reading the object's state (fig. 2, see [2] for a detailed discussion on inheritance and AOP). Its subaspects only have to define the class this aspect should be weaved to. Therefore `PersistentObject` contains an abstract pointcut `weavedClassPC()`, which has to be defined by the subaspects. We want the aspects to be instantiated for every instance of `Point` and `AnotherPoint`, so the definitions of `weavedClassPC()` in our concrete aspects corresponds to that.

But now a new problem arises: how can the state of the object be read in the pointcut method? The intention of the aspect is to be woven to classes, having the methods `getX()`, `getY()`, `setX(float)` and `setY(float)`. The set-methods are used for the pointcut definition, and the get-methods are needed by the aspect instance to read an object's state. But although knowing those method signatures the concrete type of those classes is unknown and left to those aspects, which make the abstract pointcut concrete. Because aspects crosscut the inheritance structure of classes usually those classes do not have any common type but `java.lang.Object`. So it is not possible to send getter-messages to the related object, because the type is unknown and therefore a typecast is not possible.[1] A pos-

[1] We assume here general purpose aspect languages with static type checking like AspectJ or Sally [8] which are both based on the programming language Java.

sibility would be to use reflection for those method calls, but that requires an enormous effort.

```
abstract aspect PersistentObject of
      eachobject(weavedClassPC) {
  private static int idnum = 0;
  private int id = ++idnum;
  public PersistentObject() {
    .. write new (unitialized) object to storage}
  abstract pointcut weavedInstances(Object o);
  pointcut setPC(Object o): weavedInstances(o) &&
           (receptions(void setX(float)) ||
            receptions(void setY(float)));
  after(Object p):  setPC(p) {
            ..write state to data storage}
}
```

```
aspect PersistentPoint
    extends PersistentObject {
  pointcut
   weavedInstances(Point p):
   instanceof (p);
}

aspect PersistentAnotherPoint
    extends PersistentObject {
  pointcut weavedInstances
    (AnotherPoint p):
    instanceof (p);
}
```

Figure 2: abstract persistency aspect (trial)

The concrete problem is, that aspect-oriented programming groups objects in another way than the predefined object-oriented structures do. So a mechanism is needed how to group objects of different types and allow to sent messages to them.

In the next section we discuss approaches related to this problem and demonstrate that they do not solve this problem appropriately. Afterwards we introduce and discuss *aspect-oriented adapters* for grouping types and show how this approach allows a higher level of flexibility and reduce the limitations of other approaches. We will also apply the adapter to the introducing example. In the forth section we map the introducing example to aspect-oriented adapters. Finally we summarize and conclude the paper.

2 Related Work

AspectJ offers a mechanism called *introductions* which can be applied to the given problem. The mechanism allows aspects to change the structure of the object-oriented classes. In this way additional methods and attributes can be inserted into existing classes. For the purpose of grouping objects introductions allow to insert new types to the target classes. So the interface needed by an aspect has to be defined and afterwards integrated into those classes.

Applied to the example from the first section that means, that an interface needed by the aspect `PersistentObject` has to be specified. This interface has to contain the getter-methods the aspect needs to invoke for reading a point's state. The concrete aspects `PersistentPoint` and `PersistentAnotherPoint` have to introduce this interface to the classes `Point` and `AnotherPoint`. The type of the parameter in the pointcut and pointcut method must be of that introduced interface, so the advice can invoke the getter-methods.

But this way to handle the problem leads to additional problems:

- *Tangled introduction statements*: The introduction-statements in every sub-aspect logically belong to the abstract aspect. They have to be implemented redundantly and are in that way tangled.
- *Confusing class structure*: After weaving the classes of the original application implement a new interface (fig. 3). If a lot of aspects are woven this approach leads to numerous interfaces spread all over the class structure. In this way the original code becomes confusing and makes it difficult for developers to understand the original code for reasons of reuse.
- *Lack of structure after unweaving*: After weaving developers extending the application can use the common interfaces introduced by the aspects, because they cannot distinguish the original interface from the introduced ones. So after unweaving the aspects the original classes do not implement those interfaces any longer, and so the extended application is incorrect.

Because of this we regard introductions to be inappropriate for the given problem. The problem of grouping objects has already been discussed widely in the context of object-orientation. Classes are templates from which objects are created (cf. [10]) and in that way group objects. [9] pointed out the importance of classification as a mechanism for conceptual modeling in object-oriented programming. The difference to the problem handled here is that the needed classification is not an inherent property of the objects, but depends on an aspect's subjective perspective on the system.

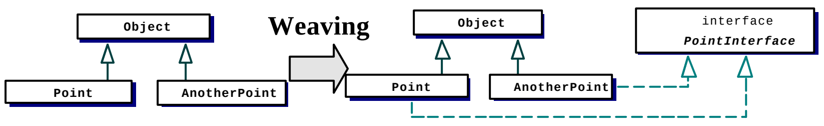

Figure 3: Using introductions for `Point` and `AnotherPoint`

In that way the mechanism of generalization introduced in [7] seems to be appropriate for the problem stated. Generalization permits to define a super-type based on an existing class. In [7] one of the main purposes of generalization is to achieve a late classification. That is exactly what aspects are doing: while they cross-cut an existing structure they accomplish a late classification for their special purposes.

Neglecting the fact, that generalization is not available in popular object-oriented programming languages, the criticism of that mechanism is corresponds to the of criticism introductions in AspectJ: developers extending the original application can not distinguish between the original classes and those created for the purpose of late classification. Also the problem of the confusing class structure stays the same.

3 Aspect-Oriented Adapters

Adapters (cf. [1]) are special classes, which adapt the interface of a class in the way expected from its clients. In that way the functionality of adapters match the problem stated above. The traditional use of adapters for mapping interfaces assumes, that clients expect a certain interface of a class which differs from that class which is able to fulfill the requests. The problem depicted here is different: advices expect their parameters to have some method signatures to send messages to them. Although the signatures are known, the type of those objects is unknown, respectively those objects do not have any common type. So an adapter is needed which has the interface expected by the aspect and which forwards messages to a certain object.

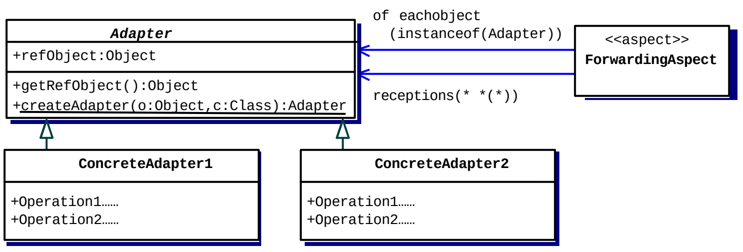

Figure 4: Aspect-Oriented Adapters[2]

An object-oriented solution for this problem is quite complex: the type of the object to which the messages have to be forwarded is unknown, so the developer has to use reflection to realized it. This code has to be used in every method which means an enormous effort.

```
public aspect ForwardingAspect
      of eachobject(instanceof(Adapter)) {
    ...
    around() returns Object: receptions(* *()) {
      ...
        return invokeMethodFromReceptionsJoinPoint((ReceptionJoinPoint) thisJoinPoint);
      ...}
    private Object invokeMethodFromReceptionsJoinPoint(ReceptionJoinPoint jp) throws Exception {
      Method m = getMethodFromReceptionJoinPoint(jp);
      return m.invoke(((Adapter) jp.getExecutingObject()).refObject, jp.getParameters());}

    private Method getMethodFromReceptionJoinPoint(ReceptionJoinPoint jp) throws Exception {
        Adapter wrap = (Adapter) jp.getExecutingObject();
        MethodSignature sig = (MethodSignature) jp.getSignature();
        String methodName = sig.getName();
        Class[] paramTypes = sig.getParameterTypes();
        return wrap.refObject.getClass().getMethod(methodName, paramTypes);}
}
```

Figure 5: Example-implementation for forward-adapter

The aspect-oriented solution for such adapters is much easier and allows a higher degree of reusability (figure 4). The abstract class `Adapter` contains the reference to the object to which every message is to be forwarded. The aspect `ForwardingAspect` is responsible for forwarding every message received by an instance of `Adapter`. Therefore an instance of `ForwardingAspect` is created for every instance of `Adapter` and the aspect contains pointcut methods, which forward every message received by the adapter to the corresponding object.

[2] The UML-like notation used here serves the understanding of the ingredients of the aspect-oriented wrapper, but does not match the UML standard.

The aspect-oriented adapter is used by creating a `ConcreteAdapter`, subclass of `Adapter`, which contains all methods needed by the aspect. Those methods have to contain a dummy implementation needed for compiling the class. The implementation will never be executed, because the `ForwardingAspect` replaces it by forward implementations.

Whenever a client wants an object to be adapted, he has to create an adapter instance by invoking the static `createAdapter(..)`-method of `Adapter`. The parameters of this method are an instance of the object which is about to be adapted, and a reference to the concrete adapter class. The adapter uses reflection to create a new instance of the concrete adapter and initializes `refObject` with the adapted object. The developer doesn't have to write glue code for forwarding messages, because this is already done by the `ForwardingAspect`.

Figure 5 shows an extract from the implementation of a forward aspect in AspectJ. The advice overrides every method of the adapter having an arbitrary return type. This is realized by a receptions pointcut consisting only of wildcards. The implementation uses the Reflection API part of AspectJ for finding out, what the target method is and the Java Reflection API for getting a reference to and invoking the target method .

For applying the aspect-oriented adapter to the introducing example a concrete adapter (`PointAdapter`) has to be created containing both getter-methods used by the advice in `PersistentObject`. The advice has to create an adapter object for the incoming object using the create method of the abstract adapters:

```
PointAdapter a = (PointAdapter) Adapter.createAdapter(p, PointAdapter.class);
```

Afterwards object a can be used as if it is an instance of `PointAdapter`.

4 Conclusion and further work

We introduced aspect-oriented adapters as a mechanism for grouping objects and compared it to existing approaches. The main advantage of using adapters is, that objects can be grouped without touching the existing inheritance structure. The effort of using aspect-oriented wrappers is compareable to introductions known from the GPAL AspectJ.

Nevertheless aspect-oriented adapters need to be used very carefully and in a disciplined maner. Because forwarding messages is realized on object-level using reflection there is no static type-checking available. So the developer has to be sure, that the interface of the adapted object really fulfills the signatures specified in the concrete adapter.

This presented approach can be used for composing *aspectual components* [6], which represent aspects whose interfaces have to be adapted to let them interact with their environment. In the future we will examine, how such components can be realized in existing general purpose aspect languages.

References

1. Gamma, E., Helm R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995
2. Hanenberg, S., Unland, R.: *Concerning AOP and Inheritance*. In: Mehner, K., Mezini, M., Pulvermüller, E., Speck, A. (Eds.): Aspect-Orientation - Workshop. Paderborn, Mai 2001, University of Paderborn, Technical Report, tr-ri-01-223, 2001
3. Hürsch, W., Lopes C.: *Separation of Concerns*. Northeastern University, technical report, no. NU-CCS-95-03, 1995.
4. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwing, J.: *Aspect-Oriented Programming*. Proceedings of ECOOP '97, LNCS 1241, Springer-Verlag, pp. 220-242, 1997
5. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: *An Overview of AspectJ*. Appears in ECOOP 2001
6. Lieberherr, K., Lorenz, D., Mezini, M.: *Programming with Aspectual Components*, Technical Report, NU-CCS-99-01, Northeastern University, Boston, 1999
7. Pedersen, C.: *Extending Ordinary Inheritance Schemes to Include Generalization*. In: Meyrowitz, N. (Ed.): Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89), October 1-6, 1989, New Orleans, Louisiana, Proceedings. SIGPLAN Notices 24(10), October 1989
8. Sally: A General-Purpose Aspect Language, `http://www.cs.uni-essen.de/dawis/research/ aop/sally/`, January 2001
9. Taivalsaari, A.: *On the Notion of Inheritance*. ACM Computing Surveys, Vol. 28, No. 3, pp. 439-479, 1996
10. Wegner, P.: *Dimensions of object-based language design*. In: Meyrowitz, N. (Ed.), Proceedings of OOPSLA '87, SIGPLAN Notices 22 (12), pp. 168-182, 1987