Refactoring a Java Code Base to AspectJ: An Illustrative Example*

Miguel P. Monteiro[1], João M. Fernandes[2]
[1]*Escola Superior de Tecnologia, IP de Castelo Branco, 6000-767 Castelo Branco PORTUGAL*
mmonteiro@di.uminho.pt
[2]*Departamento de Informática, Universidade do Minho 4710-057 Braga PORTUGAL*
jmf@di.uminho.pt

Abstract

This paper describes a refactoring process transforming a Java code base into an AspectJ equivalent. The process illustrates 17 aspect-oriented refactorings covering extraction of implementation elements to aspects, internal reorganization of extracted aspects, and extraction of commonalities to superaspects.

1. Introduction

Our aim is to expand the existing refactoring space for Aspect-Oriented Programming (AOP) [9], which is still in its infancy [5, 10, 12]. We use AspectJ [8] as the current primary representative AOP language. We base our work on the hypothesis that good-style object-oriented (OO) code can be approached as bad-style aspect-oriented (AO) code. Under this assumption, OO code betrays code smells [3], which can be removed through AO refactorings. We undertook refactoring experiments on Java code bases in order to derive interesting refactorings [12, 13]. In this paper, we illustrate results derived from implementations [6] of the Gang-of-Four (GoF) design patterns [4] in Java and AspectJ.

We present an initial validation effort for the collection of AO refactorings presented in [13] and documented in [11, 12], and to illustrate issues that arise when refactoring Java code bases to AspectJ. We describe a complete refactoring process using 17 of the refactorings. The refactoring example targets a Java implementation of Observer pattern [4], by Eckel [1]. Observer is a simple example of a crosscutting concern connecting sets of otherwise unrelated classes, implemented as a small framework.

The example also shows how the capabilities of a programming language have a profound influence on the design of programs written in that language, and even on the very idea of what comprises a good design. The starting point of the refactoring presented here is a good design in plain Java, and the final design is coded in AspectJ, which is backwards compatible to Java. Even so, the two designs are profoundly different, something that is compounded by implementation issues. The original Java implementation uses the Observable and Observer types from Java's java.util API, while the AspectJ implementation relies on internal collections owned by aspects. Consequently, the structural changes made during the refactoring process are very deep.

The refactoring process is broken in the middle into two alternative paths: (1) one performed solely in terms of the original code, and (2) another taking advantage of a reusable aspect presented in [6]. Both paths end with the same design. Space constraints prevent us to present the second path in detail, and to include the ideal number of code listings. We instead provide an eclipse project with 33 complete code snapshots. The project is available for download at www.di.uminho.pt/~jmf/PUBLI/papers/ObserverExample.zip. Throughout the process description we refer most snapshots, the same way we would do with code listings. We use code fragments to illustrate some details, and changes from the previous code state are highlighted in bold.

The rest of the paper is structured as follows. Section 2 provides specific information on the example. Section 3 describes the refactoring process. Section 4 provides a short discussion of the refactoring process and section 5 concludes this paper.

2. Design pattern Observer

The intent of Observer is to "define a one-to-many dependency between objects so that when one object

* Research partially funded by FCT under project SOFTAS (POSI/EIA/60189/2004)

changes state, all its dependents are notified and updated automatically" [4]. Observer defines the role of subject for objects generating events of interest to objects playing the role of observer. Many implementations provide subjects with an extra field for the list of its observers. Observers are added to the list by an attach operation and are removed from the list by a detach operation. When a subject gives rise to an interesting event (usually a change in state) it calls a notify operation, which in turn calls the update operation of each registered observer.

Each observer defines its reaction to a notification in the update operation. What qualifies as an interesting event is determined by the calls to update that observers make, so programmers must ensure that such calls are placed in all desired points. In very large systems, this may result in thousands of calls, scattered throughout dozens or hundreds of packages. For this reason, implementing the pattern in large systems is error–prone, and switching from one implementation to another is a hard and tedious task.

2.1. Flower example

The subject in Eckel's example [1] is a flower, whose interesting events are its two operations: open its petals and close them. These are observed by instances of two unrelated types: bees and humming birds. When the flower opens its petals, its observers have breakfast. When the flower closes its petals, its observers go to sleep. These reactions are represented by simple messages sent to the console. Each of the flower operations gives rise to a different observing relationship, as observers react differently to the two events and it is possible to support one relationship without supporting the other. The system also ensures that observers only react once to each operation. For instance, if the flower executes the open operation twice with no close in between, observers only react to the first open.

2.2. Protocol Observer-Observable

Java's java.util API provides a ready-made implementation of the Observer pattern, comprising interface Observer and class Observable. Observer classes must implement the Observer interface, which declares an update method. Subject classes must inherit from Observable, which provides the logic to manage the list of subscribed observers. Subject objects notify their observers of an interesting event by calling the notifyObservers method. In addition to the usual problems of code scattering and tangling, this solution also has the following disadvantages:

- Subject classes loose the option of inheriting from another class, as they already inherit from java.util.Observable. Observer participants are less limited because they merely implement the java.util.Observer interface, but this contributes to clutter their 'implements' clause with an interface not related to the class's primary role.
- Inheriting from java.util.Observable increases the memory footprint of each instance. Objects playing this role must carry the extra state throughout their entire life cycle, even if they only use it during certain phases.
- Use of inheritance also means that all instances will carry the extra state, even if only a subset of the instances participates in observing relationships.
- This mechanism does not support multiple separate observing relationships. If instances of a class play the subject role in various observing relationships, their observers will be notified of the events relating to all of them, and need to run extra logic to distinguish one kind of event from others.

2.3. Java implementation

Listing 1 presents Flower and listing 2 presents class Bee (Hummingbird is similar). Listing 3 shows part of the unit test. This also serves as client code.

Eckel's design partially circumvents the above limitations by relying on inner classes to isolate, within each class, the code related to the pattern. Instead of directly extending the Observer or Observable types, each participant encloses an inner class either extending Observable (subject) or implementing Observer (observers). This design has the advantage of freeing subjects to inherit from some class useful to their implementations other than java.util.Observable. It also avoids cluttering the observer's implements clause with one more interface. This design localizes within each class the code related to the pattern, but also produces an even tighter structural relationship between participants and the roles they play in the pattern. This places additional hurdles in a refactoring process aiming to replace the design.

Even Eckel's clever design cannot achieve obliviousness [2] from pattern roles. Participant classes betray the *Double Personality* smell: [13] each participant contains code related to two concerns – the primary concern and the role in the pattern. Any method of the subject (Flower) performing an interesting operation must still include code relative to its role in the pattern. In addition to this tangling, there is also code scattering: code dealing with the pattern is not modularized and each participant contains one inner class for each of the observing relationships.

There is much duplication. This is particularly noticeable in the two observers (Bee and Hummingbird), which use *four* inner classes between them. Each class duplicates the code related to the two observing relationships and each observing relationship requires a duplication of essentially the same logic.

```
public class Flower {
   private boolean isOpen;
   private OpenNotifier oNotify =
      new OpenNotifier();
   private CloseNotifier cNotify =
      new CloseNotifier();
   public Flower() {
      isOpen = false;
   }
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      isOpen = true;
      oNotify.notifyObservers();
      cNotify.open();
   }
   public void close() { // Closes its petals
      System.out.println("Flower close.");
      isOpen = false;
      cNotify.notifyObservers();
      oNotify.close();
   }
   public Observable opening() {
      return oNotify;
   }
   public Observable closing() {
      return cNotify;
   }
   private class OpenNotifier extends Observable{
      private boolean alreadyOpen = false;
      public void notifyObservers() {
         if(isOpen && !alreadyOpen) {
            setChanged();
            super.notifyObservers();
            alreadyOpen = true;
         }
      }
      public void close() {
         alreadyOpen = false;
      }
   }
   private class CloseNotifier extends Observable{
      private boolean alreadyClosed = false;
      public void notifyObservers() {
         if(!isOpen && !alreadyClosed) {
            setChanged();
            super.notifyObservers();
            alreadyClosed = true;
         }
      }
      public void open() {
         alreadyClosed = false;
      }
   }
}
```

Listing 1: Initial form of the subject class Flower.

The example includes one flower as subject, and one bee and one bird as observers. Note that each observing relationship must watch both operations, due to the requirement that observers only react to the first occurrence of an operation. Therefore, observers of open need to be notified of close, in order to determine if an open is the first to execute. The same applies to observations of close.

```
public class Bee {
   private String name;
   private OpenObserver openObsrv =
      new OpenObserver();
   private CloseObserver closeObsrv =
      new CloseObserver();
   public Bee(String nm) {
      name = nm;
   }
   // An inner class for observing openings:
   private class OpenObserver
         implements Observer {
      public void update
      (Observable ob, Object a) {
         System.out.println("Bee " + name
            + "'s breakfast time!");
      }
   }
   // Another inner class for closings:
   private class CloseObserver
         implements Observer{
      public void update
      (Observable ob, Object a) {
         System.out.println("Bee " + name
            + "'s bed time!");
      }
   }
   public Observer openObserver() {
      return openObsrv;
   }
   public Observer closeObserver() {
      return closeObsrv;
   }
}
```

Listing 2: Initial form of observer class Bee.

```
public class TestObservedFlower extends TestCase {
   Flower f = new Flower();
   Bee ba = new Bee("A"),
      bb = new Bee("B");
   Hummingbird
      hx = new Hummingbird("X"),
      hy = new Hummingbird("Y");
   public void test() {
      f.opening().addObserver(ba.openObserver());
      f.opening().addObserver(bb.openObserver());
      f.opening().addObserver(hx.openObserver());
      f.opening().addObserver(hy.openObserver());

      f.closing().addObserver(ba.closeObserver());
      f.closing().addObserver(bb.closeObserver());
      f.closing().addObserver(hx.closeObserver());
      f.closing().addObserver(hy.closeObserver());
      // Hummingbird Y decides to sleep in:
      f.opening().deleteObserver(
         hy.openObserver());
      // A change that interests observers:
      f.open();
      f.open(); // It's already open, no change.
      // Bee A doesn't want to go to bed:
      f.closing().deleteObserver(
         ba.closeObserver());
      f.close();
      f.close(); // It's already closed; no change
      f.opening().deleteObservers();
      f.open();
      f.close();
   }
```

Listing 3: Test method used throughout.

Throughout the example, an adaptation of the original test provided by Eckel is used. The test is

enhanced by two peer aspects: (1) one capturing the messages sent to the console and collating them in a string retrievable through a getter method; (2) another suppressing output to the console when the test runs in test mode and leaving it in place when it runs from the static main method. The original test does not have assertions: we added one based on the collected output.

2.4. AspectJ implementation

AspectJ solution [6] comprises an abstract aspect dealing with parts common to all cases – ObserverProtocol – and a concrete subaspect dealing with case-specific parts. The common parts are (1) the subject and observer roles, modeled by the inner (marker) interfaces Subject and Observer; (2) the maintenance of a mapping from subjects to observers, implemented with a hash table field owned by the instances of the aspect (perSubjectObservers); (3) the update logic, in which changes in the subject trigger updates in the observers. Changes in subject state are modelled by abstract pointcut subjectChange. Reactions to changes are modelled by advice acting on the joinpoints captured by subjectChange.

Parts specific to individual cases are: (1) assignment of roles subject and observer to concrete classes, implemented with 'declare parents' clauses; (2) changes on the subject that are of interest to its observers, implemented by a concrete definition of the abstract pointcut subjectChange; (3) logic to update observers at appropriate points, implemented by the updateObserver method.

Participant classes in the AspectJ implementation are completely oblivious to the pattern roles. None of the disadvantages mentioned in relation to the Java implementation applies in this case. Participant classes remain free to inherit from other classes, and instances do not expend any additional memory space when not participating in observing relationships. The mapping between a subject and its observers is maintained by the aspect itself rather than with inter-type declarations. The structure managing the mappings is defined in the abstract superaspect, so each concrete subaspect owns its own instance of this field.

3. Refactoring sessions

Note that the transformations described next follow only two of many possible paths. Though the result should always be similar, it is possible to reach it through multiple paths, since each step marks a point from which there are several possible alternatives.

Table 1 shows the refactorings. The two alternative paths start in the second phase. The first path comprises three phases, each relating to a composite refactoring [13] prescribing the use of others:

1. *Extract Feature into Aspect*: extracts the two observing relationships into aspects
2. *Tidy Up Internal Aspect Structure*: improves the internal structure of the extracted aspects
3. *Extract Superaspect*: factors out common code from the aspects to an abstract superaspect.

The second path adds ObserverProtocol early in the second phase and therefore does not use *Extract Superaspect*.

Encapsulate Implements with Declare Parents
Extend Marker Interface with Signature
Extract Feature into Aspect
Extract Inner Class to Standalone
Extract Fragment into Advice
Extract Superaspect
Generalize Target Type with Marker Interface
Inline Class within Aspect
Inline Interface within Aspect
Move Field from Class to Inter-type
Move Method from Class to Inter-type
Push Down Advice
Pull Up Marker Interface
Pull Up Pointcut
Replace Inter-type Field with Aspect Map
Replace Inter-type Method with Aspect Method
Tidy Up Internal Aspect Structure

Table 1: Refactorings used in this paper.

The eclipse project includes code snapshots presenting the code in various structural forms, always in a compilable and testable state. These are stored in the following folder hierarchy:

- *bruceeckel* – contains the code in its original form (not strictly part of the refactoring process).
- *initial* – contains the code reformatted and with a functional unit test class.
- *extractions* – contains 10 folders (named *step01–10*) showing the code at various stages during the extraction of two concerns into aspects.
- *tidyingup1* - contains 11 folders (named *step01–11*) illustrating one path to tidy up the aspect's internal structure, using the *Extract Superaspect* refactoring
- *tidyingup2* – contains 11 folders (named *step01–11*) illustrating an alternative path to tidy up the aspect's internal structure, using ObserverProtocol.

3.1. Extracting features

First phase begins with extraction of the observing relationship related to Flower.open. Three inner classes relate to this concern (see Listings 1-3 and snapshot *initial*): Flower.OpenNotifier, Bee.OpenObserver and

Hummingbird.OpenObserver. We apply *Extract Inner Class into Standalone* to Flower.OpenNotifier, yielding the following standalone class (see also snapshot *extractions.step01*):

```
public class OpenNotifier extends Observable {
   private Flower _enclosing;
   private boolean alreadyOpen = false;
   public OpenNotifier(Flower flower) {
      _enclosing = flower;
   }
   public void notifyObservers() {
      if(_enclosing.isOpen()
            && !this.alreadyOpen) {
         this.setChanged();
         super.notifyObservers();
         this.alreadyOpen = true;
      }
   }
   public void close() {
      this.alreadyOpen = false;
   }
}
```

This refactoring also entails the prior extraction of method Flower.isOpen, using *Extract Method* [3].

```
public class Flower {
   private boolean isOpen;
   private OpenNotifier oNotify =
      new OpenNotifier(this);
   //...
   boolean isOpen() {
      return isOpen;
   }
```

Next, we would like to do the same with Bee.OpenObserver and Hummingbird.OpenObserver but there are two problems. One is that each contains an action – print a message to the console – that is part of the enclosing class's primary functionality. This is dealt with by applying *Extract Method* [3] to the code fragment in each class. On Bee is as follows:

```
public class Bee {
   //...
   void breakfastTime() {
      System.out.println(
         "Bee " + name + "'s breakfast time!");
   }
   // An inner class for observing openings:
   private class OpenObserver
         implements Observer {
      public void update(Observable ob,
                         Object a) {
         breakfastTime();
      }
   }
```

The other problem is that both classes would have the same name after being turned into standalones. Since they are almost identical, it is simpler to turn them into one. However, each class holds a field referring to its enclosing class, which is of a different type. Our solution is to use *Extract Interface* [3] and use the resulting interface type instead:

```
public interface BreakfastTaker {
   public void breakfastTime();
}
```

This in turn forces us to make the breakfastTime methods public:

```
public class Bee implements BreakfastTaker {
   //...
   public void breakfastTime() {
      //...

public class Hummingbird
      implements BreakfastTaker {
   //...
   public void breakfastTime() {
      //...
```

Next, we apply *Extract Inner Class into Standalone* (see snapshot *extractions.step02*). The code is now ripe for the extraction of the various elements to an aspect. The blank aspect ObservingOpen is created and we apply the following refactorings:

- *Move Field from Class to Inter-type* to field Flower.oNotify. The private access of oNotify is (temporarily) relaxed to package-protected.
- *Move Method from Class to Inter-type* to method Flower.opening.
- *Extract Fragment into Advice* to the call to method Flower.oNotify.notifyObservers.
- *Extract Fragment into Advice* to the call to method Flower.oNotify.close.

The above refactorings move all code using the oNotify field to the aspect, so it is now possible to make it private again. The aspect now has the following contents (see *extractions.step03*):

```
public aspect ObservingOpen {
   private OpenNotifier
      Flower.oNotify =  new OpenNotifier(this);
   public Observable Flower.opening() {
      return oNotify;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning :
         flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   pointcut flowerClose(Flower flower):
      execution(void close()) && this(flower);
   after(Flower flower): flowerClose(flower) {
      flower.oNotify.close();
   }
}
```

Flower became clean of code related to the first observing relationship. The next step is to extract from observer classes Bee and Hummingbird all their remaining elements related to this concern. We apply *Move Field from Class to Inter-type* to Bee.openObsrv. This forces us to relax the field access from private to package-protected. As recommended by that refactoring, the following 'declare warning' is created:

```
   declare warning:
      get(OpenObserver Bee.openObsrv)
      && !within(ObservingOpen):
"field Bee.openObsrv accessed outside aspect.";
```

The 'declare warning' signaled an use outside the aspect of the field, in the Bee.openObserver method. This method also belongs to this concern, so we move it next, using *Move Method from Class to Inter-type*. The warnings are gone, so the 'declare warning' is removed and the access to openObsrv field is made private again. Next, similar refactorings are applied to Hummingbird. The observers are now devoid of any code related to the first observation relationship, save for the implements clause referring to BreakfastTaker (see *extractions.step04*).

The next task comprises the extraction of the second observing relationship, through a similar sequence of steps. This exposes a significant amount of duplication between the aspects, which can be factored afterwards. The following steps are:

- Apply *Extract Inner Class to Standalone* to class CloseNotifier within Flower.
- Create a new blank aspect ObservingClose.
- Apply *Move Field from Class to Inter-type* to field Flower.cNotify, whose access is temporarily relaxed from private to package-protected. This refactoring entails creating a 'declare warning' exposing 3 points in Flower still using the field.
- Apply *Move Method From Class to Inter-type* to Flower.closing, which removes one warning. The import statements in Flower can now be removed.
- Apply *Extract Fragment into Advice* to the calls to cNotify.open and cNotify.notifyObservers. This removes the two remaining warnings, so the field Flower.cNotify is made private again and the 'declare warning' is removed.

From this point on, Flower is clean of any code related to observing relationships (see *extractions.step07*). Next, we deal with the remaining code in the observer participants, Bee and Hummingbird. The first thing is to unify both CloseObserver inner classes within Bee and Hummingbird, so that *Extract Inner Class into Standalone* can be applied to both classes simultaneously, yielding a single standalone class. This entails (1) applying *Extract Method* [3] to create the bedtimeSleep method in each of them, (2) use *Extract Interface* [3] to extract BedtimeSleep. This mirrors the actions that yielded the breakfastTime method and the BreakfastTaker interface.

```
public interface BedtimeSleeper {
   public void bedtimeSleep();
}
```

Now we can use *Extract Inner Class into Standalone* to both CloseObserver inner classes to produce the following common standalone class:

```
public class CloseObserver implements Observer{
   private BedtimeSleeper _enclosing;
   public CloseObserver
         (BedtimeSleeper enclosing) {
      _enclosing = enclosing;
   }
   public void update(Observable ob, Object a) {
      _enclosing.bedtimeSleep();
   }
}
```

We then move all remaining members related to the extracted concern to the second aspect:

- Apply *Move Field From Class to Inter-type* to Bee.closeObsrv.
- Apply *Move Method From Class to Inter-type* to Bee.closeObserver.
- Apply *Move Field From Class to Inter-type* to Hummingbird.closeObsrv.
- Apply *Move Method From Class to Inter-type* to Hummingbird.closeObserver.

The import statements in Bee and Hummingbird can now be removed. The only remaining code in the participants relating to the observing relationships is the implements clauses referring to BreakfastTaker and BedtimeSleeper (see *extractions.step08*). We now use *Encapsulate Implements with Declare Parents* to both Bee and Hummingbird

```
public aspect ObservingOpen {
   declare parents: (Bee || Hummingbird)
      implements BreakfastTaker;

public aspect ObservingClose {
   declare parents: (Bee || Hummingbird)
      implements BedtimeSleeper;
```

Now all participants are completely free of any code related to extracted concerns (see *extractions.step09*).

The refactorings made until now cleaned the participant's code but it also created several standalone classes and interfaces that are used by only the aspects and provide little functionality. We therefore inline them so that all code related to observing relationships is encapsulated in the aspects. This yields code easier to reason with and to refactor.

We wanted to inline the interfaces first, but we couldn't: OpenObserver and CloseObserver depend on them. Therefore, we use *Inline Class within Aspect* on them, as well as on OpenNotifier and CloseNotifier. Next, we use *Inline Interface within Aspect* on BreakfastTaker and BedtimeSleeper. The code related to both concerns is now completely modularized within their respective aspects (see *extractions.step10* and listing 4).

3.2. Restructuring extracted aspects

As can be attested from listing 4, the internal structure of the aspects is inadequate, containing much duplication and several inner classes and interfaces

which no longer justify themselves, particularly if we want to do without the Observer/Observable API from java.util. In addition, the aspect betrays the *Aspect Laziness* smell [13]: the two aspects statically attach the additional state and behavior to the participant classes, while in this case a dynamic and unpluggable composition would be suitable. The next phase is to improve the internal structure of the aspects.

```
public aspect ObservingOpen {
   private interface BreakfastTaker {
      public void breakfastTime();
   }
   declare parents: (Bee || Hummingbird)
      implements BreakfastTaker;
   static class OpenNotifier extends Observable {
      private Flower _enclosing;
      private boolean alreadyOpen = false;
      public OpenNotifier(Flower flower) {
         _enclosing = flower;
      }
      public void notifyObservers() {
         if(_enclosing.isOpen()
               && !this.alreadyOpen) {
            this.setChanged();
            super.notifyObservers();
            this.alreadyOpen = true;
         }
      }
      public void close() {
         this.alreadyOpen = false;
      }
   }
   static class OpenObserver implements Observer {
      private BreakfastTaker _enclosing;
      public OpenObserver
            (BreakfastTaker enclosing) {
         _enclosing = enclosing;
      }
      public void update(Observable ob, Object a){
         _enclosing.breakfastTime();
      }
   }
   private OpenNotifier Flower.oNotify =
      new OpenNotifier(this);
   private OpenObserver Hummingbird.openObsrv =
      new OpenObserver(this);
   private OpenObserver Bee.openObsrv =
      new OpenObserver(this);
   public Observable Flower.opening() {
      return oNotify;
   }
   pointcut flowerOpen(Flower flower):
      execution(void open()) && this(flower);
   after(Flower flower) returning :
flowerOpen(flower) {
      flower.oNotify.notifyObservers();
   }
   pointcut flowerClose(Flower flower):
      execution(void close()) && this(flower);
   after(Flower flower): flowerClose(flower) {
      flower.oNotify.close();
   }
   public Observer Bee.openObserver() {
      return openObsrv;
   }
   public java.util.Observer
         Hummingbird.openObserver() {
      return openObsrv;
   }
}
```

Listing 4: ObservingOpen just after the extraction

Let's briefly consider options available with traditional OO. Consider a large system with a concern whose implementation is scattered throughout many classes and packages. The right approach to replace the scattered implementation would be to add a new layer abstracting its details. This would make the scattered elements easier to replace, but it would entail the patient refactoring of the system until the new layer completely hides all specific details. The refactoring process would be supported by tests targeting the new layer. Developers could develop a new implementation against the new layer's interface. Developers would leverage tests they could run against both the old implementation and the new. As soon as the new implementation is complete, it becomes possible to switch modules and rebuild the system with the new implementation. With large systems, such a process can take months.

Thanks to the modularization achieved with AOP, this duplication is now just another code smell that can be removed with further refactorings [13].

3.3. Tidying up extracted aspects

We use *Tidy Up Internal Aspect Structure* on each aspect in turn. Not only this makes their internal structures better organized, it also makes them more amenable to later apply *Extract Superaspect*, further eliminating duplication. We next show the refactoring of ObservingOpen. When the process is completed, a similar one is carried out on ObservingClose. We start by using *Generalize Target Type with Marker Interface* to eliminate duplication in inter-type declarations resulting from *Extract Feature into Aspect*. This entails creating inner marker interfaces Subject and Observer that represent pattern roles.

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}

   declare parents: Flower implements Subject;
   declare parents:
      (Bee || Hummingbird) implements Observer;
```

A name conflict arises due to two elements named Observer, which we resolve by removing the import to java.util.Observer and making all references use the full compound name. When applying *Generalize Target Type with Marker Interface* to the Flower type, we replace all references to Flower with Subject, including within inner class OpenNotifier. Since interface Subject does not 'declare method' isOpen, we use *Extend Marker Interface with Signature* on Subject to extend it with that signature. This in turn forces us to change method Flower.isOpen from package-protected to public (see *tidyingup1.step01*).

```
public aspect ObservingOpen {
   //...
   public abstract boolean Subject.isOpen();
   //...
   static class OpenNotifier
         extends java.util.Observable {
      private Subject _enclosing;
      private boolean alreadyOpen = false;

      public void notifyObservers() {
         if(_enclosing.isOpen()
               && !this.alreadyOpen) {
            this.setChanged();
            super.notifyObservers();
            this.alreadyOpen = true;
         }
      }
```

We apply *Generalize Target Type with Marker Interface* to Bee and Hummingbird, enabling us to remove BreakfastTaker and use Observer in its place. We must use *Extend Marker Interface with Signature* again, to extend Observer with the case-specific signature of breakfastTime. The step eliminates some duplication in the openObserver method, which is introduced twice, to Bee and Hummingbird. Concrete participants are now referred only in the 'declare parents' (see *tidyingup1.step02*).

We add the code related to the new implementation. When all of it is in place, we can replace the calls in the client code (i.e. the unit test) to the original implementation with calls to the new one. What follows is an elaborated variant of *Replace Inter-type Field with Aspect Map* with *Replace Inter-type Method with Aspect Method* targeting inner classes instead of inter-type fields. The step adds a mapping structure, plus associated logic (see *tidyingup1.step03*).

We add method notifyObservers, providing functionality similar to OpenNotifier.notifyObservers. notifyObservers uses a new boolean field introduced to Subject, used for the same purposes as OpenNotifier.

```
private boolean Subject.alreadyOpen = false;
private void notifyObservers(Subject subject) {
   if(subject.isOpen() && !subject.alreadyOpen){
      subject.alreadyOpen = true;
      List observers = getObservers(subject);
      for(ListIterator it =
       observers.listIterator(); it.hasNext();){
         ((Observer)it.next()).breakfastTime();
      }
   }
}
```

As prescribed in *Replace Inter-type Method with Aspect Method*, we add a 'declare warning' to expose all places where the old logic is used. The 'declare warning' targets method Subject.opening, the accessor method for the instance of inner class OpenNotifier (see *tidyingup1.step04*).

```
declare warning:
   call(java.util.Observable opening()):
   "opening() called here.";
```

Compiling again exposes six warnings, all placed in the unit test. We replace the original calls with calls to aspect logic:

```
f.opening().addObserver(ba.openObserver());
f.opening().addObserver(bb.openObserver());
f.opening().addObserver(hx.openObserver());
f.opening().addObserver(hy.openObserver());
```



```
ObservingOpen.aspectOf().addObserver(f, ba);
ObservingOpen.aspectOf().addObserver(f, bb);
ObservingOpen.aspectOf().addObserver(f, hx);
ObservingOpen.aspectOf().addObserver(f, hy);
```

The unit test now fails, due to two implementations traversing the list of observers in opposite orders. The order of notification is not relevant, so if we reverse the order with which observers are subscribed we do not really change behaviour.

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}

   public abstract boolean Subject.isOpen();
   public abstract void Observer.breakfastTime();
   private boolean Subject.alreadyOpen = false;

   private WeakHashMap subject2ObserversMap =
      new WeakHashMap();
   private List getObservers(Subject subject) {
      List observers =
         (List)subject2ObserversMap.get(subject);
      if(observers == null) {
         observers = new ArrayList();
         subject2ObserversMap.put
            (subject, observers);
      }
      return observers;
   }
   public void addObserver
         (Subject subject, Observer observer) {
      List observers = getObservers(subject);
      if(!observers.contains(observer))
         observers.add(observer);
      subject2ObserversMap.put
         (subject, observers);
   }
   public void removeObserver(Subject subject,
                              Observer observer) {
      getObservers(subject).remove(observer);
   }
   public void clearObservers(Subject subject) {
      getObservers(subject).clear();
   }
   private void notifyObservers(Subject subject) {
      //...
   }
   pointcut flowerOpen(Subject subject):
      execution(void open()) && this(subject);
   after(Subject subject) returning :
         flowerOpen(subject) {
      notifyObservers(subject);
   }
   pointcut flowerClose(Subject subject):
      execution(void close()) && this(subject);
   after(Subject subject): flowerClose(subject) {
      subject.alreadyOpen = false;
   }
   declare parents: Flower implements Subject;
   declare parents:
      (Bee || Hummingbird) implements Observer;
}
```

Listing 5: ObservingOpen aspect after tidying up.

We perform the change (see *tidyingup1.step05*) and the test now passes. After deleting code related to the original implementation, ObservingOpen is as shown in listing 5 (see also *tidyingup1.step06*).

Improving the internal structure of ObservingClose requires steps similar to those prescribed by *Tidy Up Internal Aspect Structure*, comprising the use of:

- Removal of imports of java.util.Observable and java.util.Observer. Compound names are used instead.
- Use of *Generalize Target Type with Marker Interface* requiring prior creation of private inner interfaces Observer and Subject.
- Applying *Generalize Target with Marker Interface* to Flower: references to Flower are replaced by Subject. *Extend Marker Interface with Signature* is used to introduce method isOpen to Subject.
- Applying *Generalize Target with Marker Interface* to Bee and Hummingbird, which are replaced by Observer. BedtimeSleeper is eliminated, along with the corresponding 'declare parents'. *Extend Marker Interface with Signature* is used again to introduce method bedtimeSleep to Observer.
- Use *Replace Inter-type Field with Aspect Map* with *Replace Inter-type Method with Aspect Method* to add a new implementation to ObservingClose. Following *Replace Inter-type Method with Aspect Method*, a 'declare warning' is added to expose calls to method closing.

```
declare warning:
   call(java.util.Observable closing()):
   "closing() called here.";
```

- Following the points exposed by the declare warning, the calls in the test are replaced. Again, we reverse the order in which observers are registered. We remove the declare warning and compile: the test runs successfully.

ObservingClose is now as shown in snapshot *tidyingup1.step07*.

3.4. Extracting a superaspect

Taken individually, the refactored aspects are better formed. Taken together, they betray *Duplicated Code* [3]. We eliminate the duplication by using *Extract Superaspect* to create a superaspect and pull up the common logic to it. This entails the following steps (see *tidyingup1.step10*):

- Create blank abstract aspect ObservingRelationships.
- Aspects ObservingOpen and ObservingClose are made to extend ObservingRelationships.
- *Pull Up Marker Interface* is used on Subject and Observer in both aspects, moving them to ObservingRelationships. Their access is relaxed from private to protected.
- *Pull Up Field* [3] is used on fields subject2ObserversMap in both aspects.
- *Pull Up Method* [3] is used on methods getObservers, addObserver, removeObserver and clearObservers, in both aspects.

We would like to use *Pull Up Method* [3] on method notifyObservers as well, but the method depends on many case-specific members. Thus, we merely place an abstract declaration of notifyObservers in the superaspect. Pointcuts flowerOpen and flowerClose are also case-specific and we refrain from introducing further abstract declarations. This is one of the advantages of refactoring: decisions are not set in stone. One can always change its mind later and refactor. The extracted aspect is as shown in listing 6 (see also *tidyingup1.step11*).

```
public abstract aspect ObservingRelationships {
   protected interface Subject {}
   protected interface Observer {}

   protected WeakHashMap subject2ObserversMap =
      new WeakHashMap();
   protected List getObservers(Subject subject){
      //...
   }
   public void addObserver
   (Subject subject, Observer observer) {
      //...
   }
   public void removeObserver(Subject subject,
                              Observer observer){
      getObservers(subject).remove(observer);
   }
   public void clearObservers(Subject subject) {
      getObservers(subject).clear();
   }
   protected abstract void
      notifyObservers(Subject subject);
}
```

Listing 6: Part of the extracted superaspect

3.5. Alternative refactoring path

Previous sections show how to derive an abstract aspect from existing code, but an aspect providing that functionality was available already [6]. The eclipse project therefore includes an alternative path reusing ObserverProtocol (see the 11 *tidyingup2* snapshots). It starts just after completion of the extraction process (end of section 3.1) and involves adding only case-specific parts, because ObserverProtocol already contains the general ones.

ObserverProtocol was presented as "reusable" [6], but we were forced to perform invasive changes (that is why ObserverProtocol was moved from its original package). ObserverProtocol models the events triggering the observer reactions with a *single* pointcut, but this case requires *two*. In addition, notification of

registered observers is based on a test (whether it is the first occurrence). If it succeeds, observers are notified. The test relies on field OpenNotifier.alreadyOpen. We could bind it to Subject as an inter-type declaration, but the point in the code where the test should be placed is *within* ObserverProtocol, in the advice acting on the subjectChange pointcut:

```
after(Subject s): subjectChange(s) {
   Iterator iter = getObservers(s).iterator();
   while ( iter.hasNext() ) {
      updateObserver(s,((Observer)iter.next()));
   }
}
```

This further forces us to invasively adapt ObserverProtocol. Subaspects cannot override advice inherited from superaspects, so we use *Push Down Advice* to place the advice in the subaspects, after which they are adapted. In addition, ObserverProtocol does not provide the ability to clear all observers that subscribed to a given subject. We therefore add such a method to ObserverProtocol.

4. Discussion

The refactoring process presented in this paper shows that extractions based on inter-type declarations do not change the original design, but merely modularize it. OO is a decentralized model that induces decentralized designs such as the initial Java implementation. Even after a decentralized design is modularized within an aspect, it is still a decentralized design. Once modularized, such a design may need to be changed, if not downright replaced. However, we must start by extracting it to an aspect, because many improvements can be performed only when all code is localized within a single module.

The refactoring example also shows how hard it is to obtain reusable modules, even with AOP. The abstract aspect for pattern Observer [6] had to undergo invasive changes just to be used in the simple example by Eckel [1].

5. Conclusion

This paper makes the following contributions:

- Presents a practical example of a refactoring process that goes beyond the extraction of aspects and covers the subsequent tidying up of the extracted aspects, including the internal restructurings and factoring out common code to a superaspect.
- Includes an eclipse project containing 33 complete snapshots, available as an online supplement. This project further documents the refactoring process.
- Comprises an introduction to the collection of refactorings presented in [13] and documented in [11], playing a similar role to chapter 1 of [3].
- The examples presented in this paper complement the code examples included in the description of the refactorings [11].

6. References

[1] Eckel, B., Thinking in Patterns, revision 0.9. Book in progress, May, 2003. http://64.78.49.204/ IPatterns-0.9.zip

[2] Filman, R. E., Friedman, D. P., Aspect-Oriented Programming is Quantification and Obliviousness, workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, Oct/2000.

[3] Fowler M., Beck, K., Opdyke, W. Roberts, D., Refactoring – Improving the Design of Existing Code. Addison Wesley 2000.

[4] Gamma, E.; Helm, R., Johnson, R., Vlissides, J., Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995.

[5] Hanenberg, S., Oberschulte, C., Unland, R., Refactoring of Aspect-Oriented Software, Net.ObjectDays 2003, Erfurt, Germany, Sep/2003.

[6] Hannemann, J., Kiczales, G., Design Pattern Implementation in Java and AspectJ, OOPSLA 2002, Seatle, USA, Nov/2002.

[7] Iwamoto, M., Zhao, J., Refactoring Aspect-Oriented Programs, 4th AOSD Modeling With UML Workshop, UML'2003, San Francisco, USA, Oct/2003.

[8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., An Overview of AspectJ, ECOOP 2001, Budapest, Hungary, Jun/2001.

[9] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., Aspect-Oriented Programming, ECOOP 1997, Finland, Jun/1997.

[10] Laddad, R., Aspect-Oriented Refactoring, parts 1 and 2, The Server Side, 2003. www.theserverside.com/

[11] Monteiro, M. P., Catalogue of Refactorings for AspectJ, Technical Report UM-DI-GECSD-200401, Departmento de Informática, Universidade do Minho, Aug/2004.

[12] Monteiro, M. P., Fernandes, J. M., Object-to-Aspect Refactorings for Feature Extraction, Industry track paper at AOSD'2004, Lancaster, UK, Mar/2004.

[13] Monteiro, M. P., Fernandes, J. M., Towards a Catalogue of Aspect-Oriented Refactorings, AOSD'2005, Chicago, USA, Mar/2005.