Refactoring a Java Code Base to AspectJ – An Illustrative Example

Miguel Pessoa Monteiro
Escola Superior de Tecnologia, Instit. Politécnico de Castelo Branco
Avenida do Empresário 6000-767 Castelo Branco PORTUGAL
mmonteiro@di.uminho.pt

Technical Report UM-DI-GECSD-200403
December 2004

ABSTRACT
In this report we describe a refactoring process resulting in the transformation of a small Java code base into an AspectJ equivalent. This process illustrates the use of 17 aspect-oriented refactorings covering both the extraction of implementation elements and the subsequent internal reorganisation of the resulting aspects, including the extraction of a common superaspect. Readers are provided with an eclipse project [1], available as an online suplement, presenting dozens of complete code snapshots, which further illustrate the refactoring process.

General Terms
Design, Languages.

Keywords
Aspect-Oriented Programming, Refactoring.

1. INTRODUCTION
This report is part of an ongoing effort to build a catalogue of refactorings for the AspectJ programming language. In a previous report [10] we described a refactoring experiment that enabled the characterisation of four aspect-oriented (AO) refactorings. Since then we significantly expanded the initial collection, and in this report we describe a complete refactoring process using 17 refactorings. The refactoring process described here serves both as a light introduction to the refactoring catalogue [8][11] and as a first validation effort. This report also serves to illustrate various issues that arise when refactoring Java code bases to AspectJ.

Space constraints prevent us to include the ideal number of code listings. For this reason we provide an eclipse project [9] with the complete example and dozens of code snapshots. Throughout the description of the refactoring sessions we reference many of the snapshots, the same way we would do with code listings.

The rest of this report is structured as follows. Section 2 presents the background of our work. Section 3 provides specific information on the example. Section 4 describes the refactoring process. Section 5 provides a short discussion of the refactoring process and Section 6 concludes this report.

2. BACKGROUND
Our work is based on the hypothesis that good-style object-oriented (OO) code can be approached as bad-style AO code. Using this approach, such code betrays code smells [5], which can be removed through refactorings. This motivated our adopted approach, which comprises the use of Java code bases, to be used in refactoring experiments to yield the necessary insights. In [10] we presented the results of our first case study, a framework for developing workflow applications. Though that experiment yielded some results, they were not as rich and varied as we initially hoped. For this reason we selected a different kind of code base as our second case study, the dual implementations [7] of the Gang-of-Four (GoF) design patterns [6] in both Java and AspectJ.

Table 1. Refactorings used in the refactoring process

Name of the Refactoring
Extend Marker Interface with Signature
Extract Feature into Aspect
Extract Inner Class to Standalone
Extract Fragment into Advice
Extract Superaspect
Generalise 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 Implements with Declare Parents
Replace Inter-type Field with Aspect Map
Replace Inter-type Method with Aspect Method
Tidy Up Internal Aspect Structure

The second case study yielded over twenty new refactorings, which are documented in [8]. The refactoring processes described in this report use 17 refactorings from that collection (see Table 1), three of which were presented in [10]. It is out of scope of this report to provide a detailed discussion of each refactoring. This is done in [11], where we also elaborate on code smells.

The example presented in this report is also useful to illustrate 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 session presented here is a good design in plain Java, and the final design is coded in AspectJ, which is backwards compatible with Java. Even so, these two designs are profoundly different. This is compounded by implementation issues. The original Java design uses the Observable and Observer types from Java's java.util API, while the AspectJ design relies on internal collections owned by aspects. Consequently, the structural changes made during the refactoring process are very deep.

The example is based on a Java implementation of the Observer design pattern [6], by Bruce Eckel [3]. The code is refactored into a different implementation in AspectJ. From a certain point the refactoring process is broken into two alternative paths: (1) one performed solely in terms of the original code, without the reuse of ready-made external modules and (2) another taking advantage of the reusable aspect from the AspectJ implementation by Hannemann and Kiczales [7]. The end results of both paths broadly correspond to the same design, even if the resulting code bases are not absolutely identical. Due to space constraints, only the first of these two paths is presented in detail. Interested readers will find all details in the eclipse project available online.

3. THE OBSERVER DESIGN PATTERN

The intent of the Observer design pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [6]. The pattern defines the role of **subject** for the objects generating events of interest to the objects playing the role of **observer**. Many implementations of this pattern usually provide subject objects with an extra field holding the list of its observers. Observer objects are added to the list by an attach operation and can be removed from the list by a detach operation. When the subject gives rise to one interesting event (usually a change in its state) it must call a notify operation, which traverses the list of registered observers and sends an update message to each of the observer objects.

Each observer defines its reaction to being notified in the update operation. What qualifies as an interesting event is exclusively determined by the points where the calls to the notify method are made, so programmers must ensure that such calls are made consistently and in all suitable places. In very large systems, this may result in hundreds or thousands of calls, scattered throughout dozens or hundreds of packages. For this reason, it is a hard and tedious task to switch from one implementation to another in large systems.

3.1 The Flower Example

The subject in Eckel's flower example [3] is one instance of a class representing a flower. Its interesting events are his 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 two flower operations gives rise to a different observing relationship, as the 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, the observers only react to the first open (this places an additional hurdle to the second path of refactoring process, because the reusable AspectJ implementation did not take this issue into account).

3.2 The Java Standard API Observer-Observable Protocol

Java's java.util API provides a ready-made implementation of the Observer pattern, comprising the Observer interface and the Observable class. 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.

Besides the usual problems of code scattering and tangling, this solution also has the following disadvantages:

- Subject classes loose the option of inheriting from some other class, because their "single slot of inheritance" is taken by java.util.Observable. The observer participants are less limited because they only need to implement the java.util.Observer interface. Even so, 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 play it during certain phases.
- Inheritance also means that all instances of the inheriting classes 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 in order to distinguish one kind of event from the others. It is possible to overcome this problem by making the subject pass itself as argument in the notifyObservers() method (an overloaded version of notifyObservers() that includes a parameter of type Object). However, this does not eliminate the need for some filtering logic.

3.3 The Java Implementation

Eckel's design partially circumvents the limitations stated above. It relies on inner classes to isolate, within each class, the code related to the pattern. Instead of directly extending the Observer or Observable types, each of the participants contains an inner class either extending Observable (in the case of subjects) or implementing Observer (in the case of observers). This design has the advantage of freeing subjects to inherit from some class useful to their implementations other than java.util.Observable (though the subject class does not take advantage of this in this particular example). It also avoids cluttering the observer's implements clause with one more interface. This design has the advantage of

localising, within each class, the code related to the pattern, but it also produces an even tighter structural relationship between the participants and the roles they play in the pattern. This places additional hurdles in a refactoring process aiming to switch to another design.

We next show the initial form of the participants just before the refactoring process is carried out. Listing 1 presents the subject class Flower. Listing 2 presents observer class Bee (Hummingbird is very similar). Listing 3 shows a fragment of the JUnit test [2], which in this example also doubles as client code.

```
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 Flower subject class.

What this design cannot achieve is full obliviousness [4] from the pattern roles. There is still code tangling, because each participant class contains code related to two concerns – the primary concern and the role in the pattern. Any method of the subject class (Flower) performing an interesting operation must still include code relative to the subject role. There is still code scattering, because code dealing with the pattern is not modularised. Each participant contains one inner class for each of the observing relationships. There is much duplication, which is particularly noticeable in the two observer classes (Bee and Hummingbird). The observer classes 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 code.

```
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 the Bee observer class.

```
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 as client code.

The example includes one flower as subject and one bee and one bird as observers. Note that each of the two observing relationships needs to observe both the open and close actions, though for different purposes. This is due to the requirement that an observer only reacts to the first occurrence of an operation. Therefore observers of open() need to be notified of close(), so they know if an open() is the first to execute or not. Likewise,

observations of close() must be notified of open() to know when an execution of close() is the first or not.

The example uses a JUnit test throughout[1]. It is an adapted version of the original test provided by Eckel. We added an assertion (the original test did not have any) and provided it with two peer aspects. One aspect captures the messages sent to the console and collates them in a string, which can be retrieved through a getter method. The test's assertion is based on this string. The other aspect suppresses output to the console when the test is running in unit test mode (in order to make it automatic), and leaves the messages in place when it is running in application mode (i.e. when it runs in the context of the static main method).

3.4 The Reusable AspectJ Implementation

The AspectJ design [7] comprises an abstract (and potentially reusable) aspect – ObserverProtocol – dealing with the parts common to all cases, and a concrete subaspect dealing with case-specific parts. The common parts are (1) the subject and observer roles, modelled 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 aspect (perSubjectObservers); (3) the update logic, in which changes in the subject trigger updates in the observers. This is modelled by a single abstract pointcut subjectChange() and the advice acting on its joinpoints.

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

Participant classes in the AspectJ implementation become completely oblivious to the roles they play in the pattern. None of the disadvantages mentioned in relation to the Java standard API Observer-Observable protocol applies to this implementation. The classes of participants 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 using the mechanism of inter-type declarations. The structure managing the mappings is defined in the abstract superaspect, which means that each concrete subaspect owns its own instance of this field.

4. Refactoring Sessions

It is important to note that the refactorings described in the next sections follow only a few of many possible paths. Though the final result should always be similar, it is possible to reach it through multiple paths, since each step marks a point from which it is possible to take alternative decisions.

The refactoring process presented next comprises three phases: (1) the extraction of two observing relationships into their own aspects, by applying *Extract Feature into Aspect*, (2) the improvement of their internal structures, by applying *Tidy Up Internal Aspect Structure*, and (3) factoring out of common code from the two resulting aspects, by applying *Extract Superaspect*. We present two alternative paths, starting in the second phase. In the first path, we simply tidy up the aspects and use *Extract Superaspect* to obtain the common superaspect. In the second path, we add ObserverProtocol early in the second phase and therefore *Extract Superaspect* is not used.

The eclipse project available online 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 example refactoring process).
- *initial* – contains the code reformatted and with a functional JUnit 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.

Code fragments are used to illustrate various details. Changes from the previous state are highlighted in bold.

4.1 First Phase: Feature Extraction

We start by extracting the relationship related with watching Flower.open(). There are three inner classes related with 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 entailed the prior extraction, using *Extract Method* ([5], p.110), of a getter method for Flower.isOpen:

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

[1] This test implements an use case, rather than test an individual class or method. For this reason it is adequate to regard it as a functional test rather than an unit test, even if it is implemented with JUnit.

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 should be considered part of the enclosing class's primary functionality. This is dealt with by applying *Extract Method* ([5], p.110) to the code fragment in each class. On Bee is as follows (Hummingbird is similar):

```
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 would be simpler to make them the same. However, each must hold a field referring to its enclosing class, which are of different types. Our solution is to use *Extract Interface* ([5], p.341) and use the resulting interface type instead:

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

This in turn leads 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 the Flower.opening() method.
- *Extract Fragment into Advice* to the call to the Flower.oNotify.notifyObservers() method.
- *Extract Fragment into Advice* to the call to the Flower.oNotify.close() method.

These 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 any code related to the first observing relationship. Our next step is to extract from the 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. As usual with private fields, this forces us to relax the 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 signalled an use of the field outside the aspect, 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*. There was no more warnings, so the declare warning is removed and the access to openObsrv field is turned private again. Next, similar refactorings are applied to Hummingbird. Now the observers are devoid of any code related 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 into a second aspect, through a similar sequence of steps. This exposes a significant amount of duplication between the two aspects, which can be dealt with after the concerns are modularised within the aspects. The following steps are:

- Apply *Extract Inner Class to Standalone* to the CloseNotifier class within Flower.
- Create a new blank aspect ObservingClose.
- Apply *Move Field from Class to Inter-type* to the field Flower.cNotify, whose access is temporarily relaxed from private to package-protected. Using this refactoring entailed creating a declare warning which exposed three points in Flower still using the field.
- Apply *Move Method From Class to Inter-type* to Flower.closing(). This removes one of the warnings. 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* ([5], p.110) to create the bedtimeSleep() method in each of them, (2) use *Extract Interface* ([5], p.341) to extract BedtimeSleep. This is a mirror of 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 *Replace Implements with Declare Parents* to both Bee and Hummingbird, relative to these interfaces.

```
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 provide little functionality. In addition, these are used only by the aspects. Therefore, we proceed to inline them, so that all the code related to the concerns is encapsulated in the aspects. This makes the code easier to reason with and to refactor.

We thought of inlining the interfaces first, but then we discovered we couldn't, because OpenObserver and CloseObserver depend on them. So we inline these classes first, as well as OpenNotifier and CloseNotifier, using *Inline Class within Aspect*. Next, we inline the BreakfastTaker and BedtimeSleeper interfaces, using *Inline Interface within Aspect*. All code related to the two concerns is at last completely modularised within their respective aspects (see *extractions.step10* and Listing 4).

4.2 Second Phase: Restructure the Internals of the Aspect

As can be attested from examining Listing 4, the internal structure of the aspects is confusing. It contains various duplications 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. The next phase of this refactoring process is to eliminate duplication and 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 aspect after the extraction process

Let's briefly consider the available options when working with a traditional OO system. Suppose we're dealing with a concern whose implementation code is scattered throughout many classes and packages and we would like to replace it. What would be the right approach? An adequate answer, particularly when the

system is large, would be to add a new layer abstracting the details of the existing implementation. This would make the scattered implementation 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. When the layer is in place, the developers could develop a new implementation, side-by-side with the old one, against the new layer's interface. They would leverage the tests so that they could run them 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.

Fortunately, when the concern is modularised we have more options. We can (1) continue to refactor the aspects' internals in order to transform the existing implementation into the one we want, but it is also feasible to (2) simply develop a new aspect from scratch or take an off-the-shelf solution, and plug it in the system in place of the old. This second option may not be considered a refactoring, because the refactoring concept entails the gradual transformation of the existing code until it takes the form we want for it. What is shown next is the first option.

4.2.1 Tidying Up the Aspect

It is plain is that there is much duplication in both aspects (see also *extractions.step10*). The next step is to use *Tidy Up Internal Aspect Structure* on each of them in turn. Not only will this make their internal structures better organised, it will make them more amenable to later apply *Extract Superaspect*, in order to eliminate the code duplications. We next show the refactoring of one of the aspects – ObservingOpen – and when that process is completed, a similar one is carried out on ObservingClose.

We start by using *Generalise Target Type with Marker Interface* to eliminate duplication in inter-type declarations resulting from *Extract Feature into Aspect*. This entails creating the inner marker interfaces within the aspects that represent the pattern roles – subject and observer, in this case.

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

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

This causes a name conflict because the aspect now contains two elements named Observer. We resolve this by removing the import to java.util.Observer and making all the references use the full compound name.

We first apply *Generalise Target Type with Marker Interface* to the Flower type. We replace all references to Flower with Subject, including within the inner class OpenNotifier. Since the Subject interface does not declare the isOpen() method, we use *Extend Marker Interface with Signature* on Subject to extend it with that signature. This in turn forces us to change the Flower.isOpen() method 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 next apply *Generalise Target Type with Marker Interface* to Bee and Hummingbird. This enables us to remove the inner interface BreakfastTaker, because we can now use Observer in its place. We also need to use *Extend Marker Interface with Signature* again, to extend Observer with the case-specific signature of breakfastTime(). This step also eliminated some duplication in the openObserver() method which is introduced twice (to the Bee and Hummingbird classes). The aspect now only refers to the concrete participants in the declare parents clause (see *tidyingup1.step02*).

The next step is to add the code relative to the new implementation. When all of it is in place, we can replace in the client code (in this case the JUnit test) the calls to the original implementation with calls to the new one. To some extent, what follows is an elaborated variant of *Replace Inter-type Field with Aspect Map* with *Replace Inter-type Method with Aspect Method*. The difference is that in this case they target inner classes rather than inter-type fields. This step adds a mapping structure to the aspect, as well as its associated logic (see *tidyingup1.step03*).

```
public aspect ObservingOpen {
   //...
   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();
   }
```

We also add a notifyObservers() method to the aspect, providing the same functionality as OpenNotifier.notifyObservers(). The new method uses a new boolean field that we introduce 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 exposing all places where the old logic is placed. This declare warning targets the Subject.opening() method, which is the accessor method for the instance of the 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 now replace the original calls with calls to the aspect logic:

```
public void test() {
   f.opening().addObserver(ba.openObserver());
   f.opening().addObserver(bb.openObserver());
   f.opening().addObserver(hx.openObserver());
   f.opening().addObserver(hy.openObserver());
```

```
public void test() {
   ObservingOpen.aspectOf().addObserver(f, ba);
   ObservingOpen.aspectOf().addObserver(f, bb);
   ObservingOpen.aspectOf().addObserver(f, hx);
   ObservingOpen.aspectOf().addObserver(f, hy);
```

When we run the test, the red flag appears: the test failed. After analysing the results, we discover that the problem was due to the two implementations traversing the list of observers in opposite orders. This is not an important issue in this case, so we solve the problem by reversing the order in which the observers are subscribed (see *tidyingup1.step05*):

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

We compile and test again: the test passes. Provided we do not attach importance to the order with which the observers are notified, the refactored implementation can be said to provide exactly the same behaviour as in its initial form. After deleting all the code related to the original implementation, ObservingOpen is as shown in Listing 5 (see also *tidyingup1.step06*).

4.2.2 Tidying Up the Second Aspect

Improving the internal structure of ObservingClose takes essentially the same steps as with ObservingOpen. This basically comprises the use of *Tidy Up Internal Aspect Structure*:

- Eliminate the imports to java.util.Observable and java.util.Observer, changing the code to make direct references to the complete name.
- Start applying Use *Generalise Target Type with Marker Interface*, which requires the prior creation of private inner interfaces Observer and Subject.
- Apply *Generalise Target with Marker Interface* to Flower: references to this type are replaced by Subject. *Extend Marker Interface with Signature* is used to introduce the isOpen() method to Subject, as was done with the first aspect.
- Apply *Generalise Target with Marker Interface* to Bee and Hummingbird, which are replaced by Observer. During this process the BedtimeSleeper interface is eliminated, as well as the corresponding declare parents. *Extend Marker Interface with Signature* is used again to introduce the bedtimeSleep() method to Observer.
- Use *Replace Inter-type Field with Aspect Map* with *Replace Inter-type Method with Aspect Method* to add the new implementation to ObservingClose. Following *Replace Inter-type Method with Aspect Method*, a declare warning is added to expose calls to the closing() accessor method for the CloseNotifier object:

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

- Following the places exposed by the declare warning, the calls in client (test) code are replaced. Again, we reverse the order in which the observers are registered. We remove the declare warning and compile, and the test runs successfully.

```
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) {
      if(subject.isOpen() && !subject.alreadyOpen) {
         subject.alreadyOpen = true;
         List observers = getObservers(subject);
         for(ListIterator it = observers.listIterator();
             it.hasNext();) {
            ((Observer)it.next()).breakfastTime();
         }
      }
   }
   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 being tidyied up.

After these transformations, ObservingClose is as shown in snapshot *tidyingup1.step07*.

4.3 Third Phase: Extracting Common Code to a Superaspect

Taken individually, the aspects just refactored are better formed. Together, they betray a strong dose of the *Duplicated Code* smell. As Beck and Fowler state in [5], page 76: "If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them". The next task is clear: eliminate the *Duplicated Code* smell, by eliminating the duplication in ObservingOpen and ObservingClose. This is done using *Extract Superaspect* to create a superaspect and pull the common logic up to it. This entails the following steps (see *tidyingup1.step10*):

- Create the blank abstract aspect ObservingRelationships.
- Aspects ObservingOpen and ObservingClose are made to extend ObservingRelationships.
- Use *Pull Up Marker Interface* on the Subject and Observer inner interfaces of both aspects, to move them to ObservingRelationships. Their access is relaxed from private to protected.
- Use *Pull Up Field* ([5], p.320) on the subject2ObserversMap fields of both aspects.
- Use *Pull Up Method* ([5], p.322) on the methods getObservers(), addObserver(), removeObserver() and clearObservers() of both aspects.

We would like to use *Pull Up Method* ([5], p.322) on the notifyObservers() method as well, but this depends on case-specific members in various points of the method. The notifyObservers() method from ObservingOpen uses the isOpen() and breakfastTime() methods and the alreadyOpen field, and ObservingClose has similar dependencies. For this reason we just leave an abstract declaration of notifyObservers() in the superaspect.

We also have the option of using *Pull Up Pointcut* to the flowerOpen() and flowerClose() pointcuts, but they are case-specific. We should at most place abstract declarations in the superaspect. We refrain from doing that at this stage. This is one of the advantages of refactoring: decisions are not set in stone, and one can always change its mind later, and then refactor. So, the aspects take the final forms as shown in Listing 6 and Listing 7 (see also *tidyingup1.step11*).

4.4 Alternative Refactoring Path

The previous sections show how to use refactoring to gradually transform an implementation in Java into an implementation in AspectJ. However, a reusable aspect providing the same functionality [7] – ObserverProtocol – is already available. It is reasonable to ask whether would it be possible to take advantage of this aspect while refactoring the aspects. The eclipse project available online includes an alternative path proving that such a refactoring can indeed be accomplished (see the 11 *tidyingup2* snapshots). This second path starts at the point at which the extraction process was completed (end of section 4.1). This refactoring process involves adding fewer amounts of new code compared to the previous one, because the framework is already set by ObserverProtocol. Only the concrete parts are missing.

In [7] ObserverProtocol is referred as "reusable". The second path provided an opportunity to test the extent to which it proves to be so. As illustrated in the online project, most of its code could indeed be reused for the example at hand, but we nevertheless felt the need to perform invasive changes in the code of ObserverProtocol. For this reason we removed ObserverProtocol from its original package and placed it in the package used in the example.

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

   protected WeakHashMap subject2ObserversMap =
      new WeakHashMap();
   protected 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();
   }
   protected abstract void
      notifyObservers(Subject subject);
}
```

Listing 6. The ObservingRelationships extracted superaspect

```
public aspect ObservingOpen
      extends ObservingRelationships {
   public abstract boolean Subject.isOpen();
   public abstract void Observer.breakfastTime();
   private boolean Subject.alreadyOpen = false;

   protected 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();
         }
      }
   }
   private pointcut flowerOpen(Subject subject):
      execution(void open()) && this(subject);
   after(Subject subject) returning :
         flowerOpen(subject) {
      notifyObservers(subject);
   }
   protected 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 7. ObservingOpen aspect

The primary cause for the invasive changes is as follows. ObserverProtocol expects the events triggering the reactions of the observers to be represented by a single pointcut –

subjectChange() – but in this particular case, it is convenient to use two. In addition, this case requires that notification of all observers in the subject's list depend on the result of a test (if it is the first time that the event occurs). Only if the test succeeds are the subject's registered observers notified. This test relies on the alreadyOpen field of the OpenNotifier inner class. This functionality can be obtained by moving that field to Subject, as an inter-type declaration. The problem is that the point in the code where that test should be placed is within ObserverProtocol, in the advice acting on the subjectChange() pointcut:

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

This forces us to modify ObserverProtocol, so that it can be used in this example. Since a subaspect cannot override the advice inherited from its superaspect, the advice itself must be pushed down from ObserverProtocol to the subaspects (using *Push Down Advice*) and then subject to case-specific changes.

In addition, there was a piece of (reusable) functionality missing in ObserverProtocol: the ability to clear all observers that subscribed to a given subject. We therefore added to ObserverProtocol the following method:

```
public abstract aspect ObserverProtocol {
   //...
   public void clearObservers(Subject s) {
      getObservers(s).clear();
   }
```

5. DISCUSSION

The refactoring process presented in this report shows that extractions based on inter-type declarations do not change the original design, they merely modularise it. OO is a decentralised model, and leads to decentralised designs such as the initial implementation of Observer. As the refactoring example shows, when a decentralised design is modularised within an aspect, the result is still a modularised and decentralised design, not a centralised one. Such a design still needs to be improved, if not downright replaced with something more suitable. Yet it still makes sense to first modularise the code, because improvements are easier to achieve when we can benefit from the advantages of modularity.

The refactoring example also shows how hard is to achieve reusable modules, even with aspect-oriented programming. The abstract aspect for the Observer pattern [7] had to undergo invasive changes just to be used in the simple example by Eckel [3]. We nevertheless believe that this aspect has the potential to be further developed in order to encompass a wider variety of situations, though that will almost certainly come at the cost of making its design more complex.

6. CONCLUSION

This report makes the following contributions:

- It presents a practical example of a refactoring process, using a new collection of aspect-oriented refactorings. The refactoring process does not limit itself to extractions of implementation elements, and covers the subsequent tidying up of the resulting aspects, including the internal restructurings and the extraction of a common superaspect.
- It includes an eclipse project containing dozens of complete code snapshots, available as an online suplement. This project further documents the refactoring process.
- It comprises an introduction to the collection of refactorings presented in [8], playing an introductory role similar to chapter 1 of [5].
- The examples presented in this report complement the code examples included in the refactorings documented in [8].

7. ACKNOWLEDGMENTS

Miguel Pessoa Monteiro is partially supported by *PRODEP III (Medida 5 – Acção 5.3 – Eixo 3 – Formação Avançada de Docentes do Ensino Superior)*.

8. REFERENCES

[1] Eclipse home page. http://www.eclipse.org/

[2] JUnit home page. http://www.junit.org/

[3] Eckel, B., Thinking in Patterns, revision 0.9, Book In progress, May 20, 2003. Available at http://64.78.49.204/ IPatterns-0.9.zip

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

[5] Fowler, M. (with contributions by K. Beck, W. Opdyke and D. Roberts), Refactoring – Improving the Design of Existing Code, Addison Wesley 2000. ISBN 0201485672.

[6] Gamma, E., Helm, R., Johnson, R. Vlissides, J., Design Patterns, Elements of Reusable Object-Oriented Software, Addison Wesley, 1995. ISBN 0201633612.

[7] Hannemann, J. Kiczales, G., Design Pattern Implementation in Java and AspectJ, OOPSLA 2002, November 2002.

[8] Monteiro, M. P., Catalogue of Refactorings for AspectJ, Technical Report UM-DI-GECSD-200402, Departmento de Informática, Universidade do Minho, December 2004. Available at http://www.di.uminho.pt/~jmf/PUBLI/papers/ 2004-TR-02.pdf

[9] Monteiro, M. P., Observer Example – eclipse project available at http://www.di.uminho.pt/~jmf/PUBLI/papers/ ObserverExample.zip

[10] Monteiro, M. P., Fernandes, J. M., Object-to-Aspect Refactorings for Feature Extraction, industry paper presented at the AOSD'2004, UK, Lancaster, March 2004. Available at http://aosd.net/ 2004/archive/Monteiro.pdf.

[11] Monteiro, M. P., Fernandes, J. M., Towards a Catalogue of Aspect-Oriented Refactorings, to be published in the proceedings of the AOSD 2005, Chicago, USA, March 2005.