SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2008; **38**:361–396
Published online 21 August 2007 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.835

An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms

Miguel P. Monteiro[1] and João M. Fernandes[2,*,†]

[1]*Departamento de Informática, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2829-516 Caparica, Portugal*
[2]*Departamento de Informática & CCTC, Universidade do Minho, 4710-057 Braga, Portugal*

SUMMARY

This paper describes a refactoring process that transforms a Java source code base into a functionally equivalent AspectJ source code base. The process illustrates the use of a collection of refactorings for aspect-oriented source code, covering the extraction of scattered implementation elements to aspects, the internal reorganization of the extracted aspects and the extraction of commonalities to super-aspects. Copyright © 2007 John Wiley & Sons, Ltd.

Received 12 May 2006; Revised 29 April 2007; Accepted 1 May 2007

KEY WORDS: refactoring; aspect-oriented programming; design patterns

1. INTRODUCTION

Refactoring [1] and aspect-oriented programming (AOP) [2] are two techniques that contribute to deal with the problems of continuous evolution of software. This paper shows how both techniques can be combined to improve the structure of an existing system. In doing so, it also illustrates current notions of good style for AOP.

*Correspondence to: João M. Fernandes, Departamento de Informática, Universidade do Minho, 4710-057 Braga, Portugal.
†E-mail: jmf@di.uminho.pt

Contract/grant sponsor: FCT
Contract/grant sponsor: FEDER; contract/grant number: POSC/EIA/60189/2004

AOP enables the modularization of crosscutting concerns (CCCs), thus diminishing the potential impact of changes on CCCs to code not directly related to such concerns. The advent of AOP brings forward the problem of how to deal with large number of existing object-oriented (OO) code bases. Experience with refactoring of OO software in recent years suggests that refactoring techniques can be effective in bringing concepts and mechanisms of aspect orientation to existing OO applications and frameworks. In previous work [3,4], we sought to expand the existing refactoring space for AOP, which is currently under development [5–7]. This work is based on AspectJ [8], a backwards-compatible extension to Java that supports the mechanisms of AOP. AspectJ is currently the primary representative of AOP at the level of programming languages. We undertook refactoring experiments on code bases in Java and/or AspectJ to derive interesting refactorings [3,4,9].

A consequence of AspectJ's backwards compatibility with Java is that AspectJ supports multiple programming styles: those of Java and those that are based on aspect-specific mechanisms. In [4], we argue that Java's traditional OO ways to deal with CCCs should be considered bad style as more effective AspectJ-like mechanisms are available. In this light, traditional OO ways to deal with CCCs should be regarded as bad style, amenable to improvement through a process of refactoring.

In this paper, we illustrate results derived from experiments [4,9] performed on existing implementations of the Gang-of-Four (GoF) design patterns [10], coded in Java [11,12] and AspectJ [13]. To this end, the paper describes a complete refactoring process using 17 of the new refactorings documented in [9]. The process targets a Java implementation of the *Observer* design pattern [10], implemented in Java by Eckel [11]. The aim is to transform Eckel's implementation into the AspectJ implementation described in [13]. This process was originally characterized in the context of a validation effort for the collection of aspect-oriented (AO) refactorings documented in [4,9].

The refactoring process described in this paper also illustrates how the capabilities of a programming language have a strong 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 process is an example of good Java design, created by the author of a popular Java tutorial [14]. The finishing point is generally regarded by the AspectJ community as an example of good design [13,15]. Nevertheless, the two designs are profoundly different. In addition, the Java implementation uses the `Observable` and `Observer` types from Java's `java.util` API, while the AspectJ implementation [13] relies on an internal mapping structure owned by the aspect modules. Consequently, the structural changes made during the refactoring process are very deep.

Several authors noted that AspectJ pointcuts can easily break when the code base to which they compose is modified [5,6,16], a problem that is currently known as the *fragile pointcut* problem [17]. As a consequence, many traditional OO refactorings can be unsound in the presence of aspects. Though the process described in this paper does use several OO refactorings from [1], the *fragile pointcut* problem is out of the scope of the paper and is not tackled here. We refer interested readers to [18,19].

This paper is a revised and extended version of a paper presented at ICSM'05 [20]. The main additional contributions relative to the other paper are the more thorough analysis of the Observer pattern and the greater detail of the descriptions of the refactoring processes, particularly the second, alternative path (Section 6). This paper also includes a more developed discussion section and a survey of related work. Though this paper was written to stand on its own, an eclipse project complementing the paper is available online[‡].

The rest of the paper is structured as follows. Sections 2 and 3 present brief introductions to refactoring and AOP, respectively. Section 4 provides specific information on the system used as target of refactoring process. Section 5 describes the refactoring process. Section 6 describes a second refactoring process that is based on an abstract aspect from [13], comprising an alternative to the path described in the previous section. Section 7 provides a discussion and Section 8 surveys related work. Section 9 concludes the paper.

2. REFACTORING

Refactoring [1,21,22] is a technique that aims to improve the internal structure of a software system, at the level of the source code, without changing its externally observable behaviour. A refactoring process comprises a sequence of small behaviour-preserving transformations of source code, also called refactorings. Each individual refactoring should be small to better ensure safety, but a process comprising a sequence of refactorings can yield a profound effect on the structure of a software system. Refactoring can be useful to evolve software in line with changes in environments and requirements.

Programmers have been performing *ad hoc* behaviour-preserving transformations for decades, though they did not call it refactoring. Only at the start of 1990s it did become the subject of formal study. The earliest works were by Opdyke and Johnson [23], who first coined the word *refactoring* and focused on OO frameworks, and by Griswold and Notkin [24], who focused on block-structured imperative programs and functional programs. Refactoring became widely known after the book by Fowler *et al.* [1] was published at the end of the 1990s. One important contribution of the book is to express notions of good style for OO source code, through collections of *code smells*, i.e. symptoms in source code that are indicative of opportunities to improve the structure through refactoring.

Fowler's book uses Java as the subject language and was published at a time when tool support for refactoring Java programs was not available. Tool developers responded positively to Fowler's challenge, and thanks to that present users of various integrated development environments can benefit from automated support for many of the refactorings described in Fowler's book. Beyond that, the concept of refactoring remains an effective way to express notions of style, as can be attested in the refactoring workbook [25] and the many exchanges in the mailing list[§] dedicated to the topic. The focus of this paper is centred on this approach to refactoring, applied to AOP source code.

Fowler *et al.* [1] noted that coverage by a set of unit tests is a prerequisite for the use of refactoring, in order to ensure that the behaviour is not changed. Likewise, the process described in Sections 5 and 6 of the paper is supported by an adapted version of a unit test originally found in Eckel's example (not shown in the paper: it can be found in the eclipse project available online). In addition,

[‡]The project can be downloaded from http://www.di.uminho.pt/~jmf/papers/ObserverExample.zip. It includes 33 complete code snapshots. The 'reusable' aspect from [13] is not placed in its original package, as the refactoring process required invasive changes on it. For that reason, the aspect is placed in the same package as the other elements of the system being refactored.

[§]http://tech.groups.yahoo.com/group/refactoring/.

Laddad [7] proposed taking the advantage of the visualization capabilities of the AJDT plug-in for eclipse, namely the gutter annotations that show the interactions between aspects and classes. These can be checked before and after performing a refactoring. In addition, some aspect-specific techniques are proposed in [7].

3. ASPECT-ORIENTED PROGRAMMING

In OO systems, it often happens that certain kinds of concern, such as persistence, exception handling, logging and distribution, are scattered across the units of modularity of the system. Traditional OO mechanisms are unable to localize the code related to such concerns within a single module. Consequently, the representation of such concerns takes the form of multiple, small code fragments that are scattered throughout the classes of the system, a phenomenon usually referred as *code scattering*. Kiczales *et al.* [2] refer to the concerns that give rise to code scattering as CCCs. In addition, the various code fragments related to CCCs tend to be mixed with the code related to the primary functionality of the system's existing modules, harming the comprehensibility and ease of evolution of all concerns involved. This negative effect is dubbed *code tangling* by Kiczales *et al.* [2]. The implementation of a number of design patterns are examples of CCCs [13], including *Observer* [10], the pattern surveyed in Section 4.

AOP [2,26,27] is a new programming paradigm providing constructs explicitly devoted to localize source code-related CCCs in their own units of modularity—called *aspects* [2]—thus, eliminating code scattering and tangling. Currently, the most mature AOP language is AspectJ [8,28–30], an extension to Java that supports AOP. In the remainder of this section we briefly describe some of the novel mechanisms of AspectJ that are used in the refactoring process described in the paper.

In AspectJ, aspects are class-like modules that can hold state and behaviour and are provided with novel mechanisms through which aspects compose their functionality to multiple, scattered points of a given system. The most important mechanisms are based on the concept of *joinpoint*, i.e. events that occur during the execution of a program, namely method calls, constructor executions and accesses to instance fields. AspectJ provides a construct called *pointcut* able to capture a set of joinpoints related to non-contiguous points in the source code base. For instance, Listing 1 shows a pointcut capturing all calls to the public methods of java.io.PrintStream having any number of arguments, void as return type, and a name starting with 'print'.

In the example, the *pointcut designator* (PCD) call is used to capture the methods calls of interest. AspectJ provides a rich set of PCDs that includes a number of PCDs that restrict the set of captured joinpoints. In addition, PCDs can also be composed like predicates, using operators &&, || and !, which express set union, set intersection and set complement respectively. Listing 2 shows a pointcut similar to that of Listing 1, but complemented with a within PCD that restricts captured calls to those that originate within the lexical boundary of class Capsule.

Aspects also include *advice*, i.e. nameless block-statement constructs that implicitly execute upon the occurrence of each specified joinpoint. Advice can run before, after or instead of each captured

```
public pointcut allCalls2SystemOutPrints(): call(public void java.io.PrintStream.print*(..));
```

Listing 1. Example of an Aspect pointcut.

```
public pointcut callsFromCapsule2SystemOutPrints():
   call(public void java.io.PrintStream.print*(..)) &&
   within(Capsule);
```

Listing 2. Example of a composition of two pointcut designators.

```
   void around(): allCalls2SystemOutPrints() {
      System.out.println("message printed.");
   }
```

Listing 3. Example of an Aspect around advice.

```
  public pointcut messagesFromSystemOutPrint(Object message):
     allCalls2SystemOutPrints() && args(message);

  void around(Object message): messagesFromSystemOutPrint(message) {
     String newMessage = "[" + message.toString() + "]";
     proceed(newMessage);
 }
```

Listing 4. Example of a pointcut capturing corner from the jointpoints.

```
   private boolean Server.disabled = false;
```

Listing 5. Example of an inter-type declaration.

joinpoint. The latter are called *around advice*. Listing 3 shows a piece of around advice that executes upon each method call captured.

The instructions within the advice from Listing 3 are executed instead of the captured joinpoints. This ability to execute blocks of statements instead of the original joinpoints (as in Listing 3) means that aspects can delete, or at least circumvent, the existing behaviour of a target code base. AspectJ also provides the keyword *proceed* through which around advice can execute the original, captured joinpoint. In addition, pointcuts can also capture context data from the captured joinpoints. For instance, Listing 4 shows a pointcut similar to that of Listing 1 that also captures the argument to the print method. In doing so, it also restricts the set of captured joinpoints to those calls that receive one argument of the specified type. The around advice from Listing 4 is implicitly called upon each of the joinpoints captured by the new pointcut and binds the method argument to variable message. Thus, the advice shown in Listing 4 adds square brackets to the beginning and end of the messages sent to the console.

The above composition mechanisms are often dubbed *dynamic crosscutting*. In addition, aspects also provide *static crosscutting* in the form of *inter-type declarations*, i.e. the ability of aspects to *introduce* additional state and behaviour to a set of target classes. Though the declarations are placed within the aspects at the source level, the target classes are the owners of the introduced members at the binary level. For instance, the inter-type declaration in Listing 5 declares that every instance of class Server has additional field *disabled*, of type boolean, initialized to false. Similar declarations can be made of methods. The visibility of inter-type members is relative to the aspect, not to target classes. When an aspect declares an inter-type member as private, only code within the aspect

```
declare warning: allCalls2SystemOutPrints() &&
   !within(DeclaringAspect): "Do not use System.out.print*().";
```

Listing 6. Example of a declare warning.

can use those members, despite the fact that the target classes are the owners. This further ensures the modularity of the concern related to the aspect.

In addition, AspectJ provides constructs through which the compiler is configured to generate additional error and warning messages. The constructs are clauses declare warning and declare error and accept a pointcut stating the joinpoints that give rise to the warning or error, and the associated message. For example, Listing 6 shows a declare warning that reuses the pointcut shown in Listing 1 and generates a warning upon every call to methods from System.out whose name starts with ‘print’ and that are not issued from the module named DeclaringAspect.

Finally, AspectJ provides the declare parents construct through which classes are made to implement additional interfaces or extend a different superclass, subject to restrictions derived from the type rules of Java. A popular idiom based on the use of interfaces comprises declaring a *marker interface* that represents an abstract role. In many cases the marker interface can be declared within the aspect as an inner interface, often with private visibility, ensuring that no code outside the aspect depends on the role represented by the interface. On this basis, the aspect uses inter-type declarations to add state and behaviour to the interface and uses a declare parents clause to make concrete classes implement it. Thus, the aspect composes extra functionality to a number of target classes in a transparent way. In some cases, it is feasible to separate the case-specific parts of the aspect code from the generally applicable parts, in which case an abstract base aspect can be extracted that is potentially reusable. This design technique is widely used in the examples presented in [13], including the AspectJ implementation of *Observer* that we describe in detail in Section 4.4.

4. THE OBSERVER DESIGN PATTERN

The intent of *Observer* (also known as *Publish-Subscribe*) 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’ [10]. The pattern is an example of a CCC connecting sets of otherwise unrelated classes, implemented as a simple framework. *Observer* defines the role of *subject* for objects generating events of interest to other objects, which play the role of *observer*. Many implementations provide subjects with an extra field holding the list of their registered observers. Observers are registered (i.e. 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 its state—it calls a *notify* operation, which in turn calls the *update* operation of each registered observer. The general structure of *Observer* is presented in Figure 1, which is similar to that used in [10].

The example in the GoF book [10] is coded in C++ and is therefore based on abstract classes to represent the roles of subject and observer. With Java, it is usual to represent this kind of role with interfaces. In each case, the implementing classes are called *concrete subject* and *concrete observer*, respectively.

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 subjects make, so programmers must

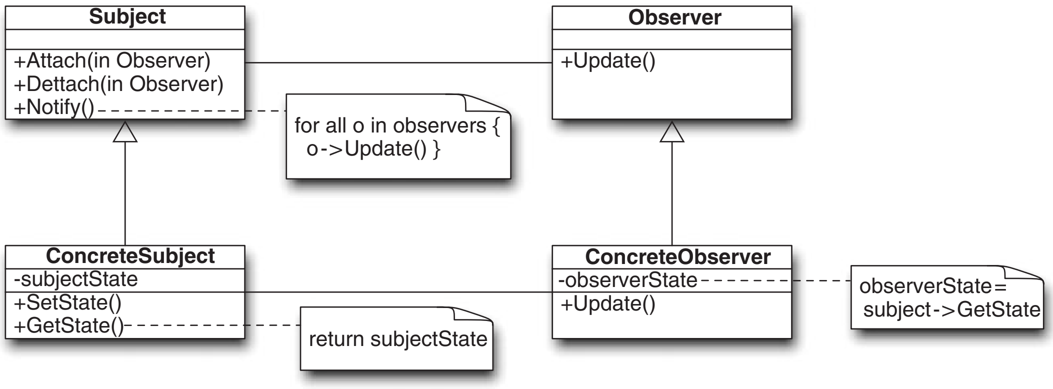

Figure 1. General structure of the observer design pattern.

ensure that such calls are placed in all desired points. In large systems, this may result in many calls, scattered throughout lots of packages. This is probably one of the reasons why implementing the pattern in large systems can be error prone [31]. Switching from one scattered implementation to another in large systems is a tedious and risky task.

4.1. The Flower example of Observer

The subject in Eckel's example [11] is one instance of a class representing a *flower*. Its interesting events are the two operations it provides: open its petals and close them. These are observed by one instance each 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. Eckel's example also ensures that observers only react once to each operation, i.e. if the flower executes the *open* operation twice with no *close* in between, observers are notified only upon the first *open*. Note that the order with which observers are notified is not relevant[¶].

4.2. The Java standard Observer-Observable API

The java.util API of Java provides a ready-made implementation of *Observer*, comprising interface Observer and class Observable. Observer classes must implement Observer, which declares an update method. Subject classes must inherit from Observable, which provides the logic to manage the list of registered observers. Subject objects notify their observers of an interesting event by calling method notifyObservers, which is overloaded with the following

[¶]The javadoc documentation for method Observable.addObserver(), used by Eckel in his implementation, states that 'the order in which notifications will be delivered to multiple observers is not specified'.

signatures:

```
public void notifyObservers(Object arg)
public void notifyObservers()         // equivalent to notifyObservers(null)
```

This method only notifies the registered observers if the object was previously marked as having been changed, by executing the setChanged method. Observer declares method update, with the following signature:

```
void update(Observable o, Object arg)
```

Subjects can use the second parameter of type Object to pass data to its observers. In order to be general purpose, the parameter must accept any type, which in Java means java.lang.Object. This provides the necessary flexibility but has the disadvantage of placing this parameter outside the reach of the type checker[‖]. It is the programmer's responsibility to ensure that subjects and its observers use the runtime types consistently. Method update contains the actions to be carried out by observer objects when they are notified of an interesting event. What qualifies as 'interesting' is exclusively determined by the places where calls to notify are made. Programmers using the protocol must ensure that such calls are made in all suitable places. This includes calling method setChanged. In large systems, this may comprise a large number of calls, scattered throughout lots of packages. Naturally, this has the consequence that it would be hard to switch to a different implementation after the structure is in place (for instance, one that does not require calls to a setChanged() method).

In addition to the usual problems of code scattering and tangling [2], this solution 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 interface java.util.Observer, but this contributes to clutter their implements clause with an interface not related to the primary role of the class.
- 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.
- The mechanism does not provide appropriate support for 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. It is possible to ameliorate this problem by making the subject pass itself as argument in the version of notifyObservers with two arguments, but this merely pushes the filtering logic from subjects to observers.

[‖]Java 5 supports generic types but the Observer/Observable protocol was not updated to support generics.
[**]It is fair to note that this limitation does not apply to all possible Java implementations of *Observer*. The implementation described in Section 4.3 manages to circumvent it.

4.3. The original pure Java implementation

Listing 7 presents class Flower and Listing 8 presents class Bee (Hummingbird is similar) as written by Eckel [11]. Eckel's design partially circumvents the limitations mentioned in the previous section. The design 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 participant encloses an inner class either extending Observable or implementing Observer. Each participant contains one inner class for each of the observing relationships.

Inner classes provide users of Java a limited form of multiple inheritance, in that inner classes can refer to the members of its enclosing class, even private ones. Inner classes are still free to

```
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{
     // similar to OpenNotifier, but focusing on operation close.
 }
}
```

Listing 7. Initial form of the subject class Flower, as presented in [11].

inherit from some other class, so they enable the enclosing class to make a part of itself inherit from a given class while remaining free to use the more traditional use of single inheritance. Eckel uses this mechanism in both the subject and the observer participants, with the benefit of giving subjects the option to inherit from some useful class other than java.util.Observable (though this particular example does not take advantage of this). It also avoids cluttering the observer's implements clause with an additional interface. The design manages to localize, within each class, the code related to the pattern, but nevertheless it results in a tight structural relationship between participants and the roles they play in the pattern. It should be noted that the use of inner classes requires additional refactoring steps, as shown in Section 5.1.

What this clever design cannot achieve is obliviousness [32] from pattern roles. Participant classes betray the *Double Personality* smell [4], i.e. each participant plays more than one role in the system and therefore contains code related to more than one concern. This is a form of code tangling. Any method of the subject—Flower (Listing 7)—performing an interesting operation must still include code related to its role in the pattern. 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 duplication (i.e. the *Duplicated Code* smell [1]), which is particularly noticeable in the two observers—Bee (Listing 8) 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.

The instance of Flower plays the role of subject. Instances of Bee and Hummingbird are observers. Each observing relationship must monitor both operations, due to the requirement that observers only react to the first of multiple consecutive occurrences of the same operation (as is the case with the unit test from Eckel's example [11]). Therefore, observers of *open* need to be notified of *close*, to determine whether a call to *open* belongs to a sequence of calls to *open* without calls to *close* in between. This applies to observations of both *open* and *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 8. Initial form of observer class Bee.

4.4. The reusable AO implementation of Observer

The AspectJ implementation proposed in [13] comprises an abstract base aspect—ObserverProtocol (Listing 9)—that deals with the parts common to all cases, plus concrete sub-aspects that deal with case-specific parts (Listing 10). The common parts comprise the following:

- The subject and observer roles, modelled by the inner marker interfaces Subject and Observer.
- Maintenance of a mapping from subjects to observers, implemented with a hash table field (perSubjectObservers), owned by instances of the aspect.
- The *update* logic, in which changes in the subject trigger updates in the observers. Changes in subject state are modelled by abstract pointcut subjectChange.

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

  private WeakHashMap perSubjectObservers;

  protected List getObservers(Subject subject) {
     if (perSubjectObservers == null) {
        perSubjectObservers = new WeakHashMap();
     }
     List observers = (List)perSubjectObservers.get(subject);
     if ( observers == null ) {
        observers = new LinkedList();
        perSubjectObservers.put(subject, observers);
     }
     return observers;
  }

  public void addObserver(Subject subject, Observer observer) {
     getObservers(subject).add(observer);
  }

  public void removeObserver(Subject subject, Observer observer) {
     getObservers(subject).remove(observer);
  }

  protected abstract pointcut subjectChange(Subject s);

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

  protected abstract void updateObserver(Subject subject, Observer observer);

}
```

Listing 9. Reusable aspect implementation of *Observer* from [13].

```
public aspect ColourObserver extends ObserverProtocol{
   declare parents: Point implements Subject;
   declare parents: Screen implements Observer;

   protected pointcut subjectChange(Subject subject):
      call(void Point.setColour(Colour)) && target(subject);

   protected void updateObserver(Subject subject, Observer o) {
      ((Screen)o).display("Screen updated " + "(point subject changed colour).");
   }
}
```

Listing 10. Example of case specific, concrete aspect reusing ObserverProtocol.

Parts specific to individual cases are:

- Assignment of roles subject and observer to concrete classes implemented through declare parents clauses.
- Changes to the subject that are of interest to its observers implemented by a definition of abstract pointcut subjectChange.
- The logic to update observers at appropriate points, implemented by aspect method updateObserver.

Participant classes in the AspectJ implementation are oblivious to the pattern roles. None of the disadvantages mentioned in relation to the Java implementation applies to this case. Participant classes remain free to inherit from other classes, and instances do not expend additional memory when not participating in observing relationships. Note that the mapping between a subject and its observers is maintained by the aspect itself, rather than being supported by means of inter-type declarations, as it could be expected from a more straightforward implementation: if the aspect resorted to inter-type declarations, all instances of the target classes would be affected, throughout their entire life cycles. The structure managing the mappings is defined in ObserverProtocol, so each concrete sub-aspect has its own instance of the hash map. One disadvantage of this implementation is a possible degradation in performance, when dealing with large systems comprising large numbers of participant objects (no performance measurement is presented in [13]).

5. THE REFACTORING PROCESS

Tables I–III present the 17 refactorings from [4,9] that are used throughout the refactoring process. As with most collections of refactorings, each individual refactoring should be small to better ensure safety. The process illustrated in this section is carried out according to the strategy proposed in [4,9] for the extraction of a CCC. The strategy establishes that prior to anything else, all elements related to the target concern should be moved to a single module, using refactorings from Table I. After the extraction is completed, the focus should be on improving the internal structure of the extracted aspects, using the refactorings from Table II. The kind of improvements we describe are considerably easier to perform (or possible at all) after the associated implementation is modularized.

Table I. Refactorings to extract crosscutting concerns into aspects.

Name	Typical situation	Recommended action
Extract Feature into Aspect	Code related to a feature is scattered across multiple methods and classes, tangled with unrelated code	Extract to an aspect all implementation elements related to the feature
Extract Fragment into Advice	Part of a method is related to a concern whose code is being moved to an aspect	Create a pointcut capturing the appropriate joinpoint and context and move the code fragment to an advice based on the pointcut
Extract Inner Class to Standalone	An inner class relates to a concern being extracted into an aspect	Eliminate dependencies from the enclosing class and turn the inner class into a stand-alone class
Inline Interface within Aspect	One or several interfaces are used only by an aspect	Move the interfaces to within the aspect
Move Field from Class to Inter-type	A field relates to a concern other than the primary concern of its owner class	Move the field from the class to the aspect as an inter-type declaration
Move Method from Class to Inter-type	A method belongs to a concern other than the primary concern of its owner class	Move the method into the aspect that encapsulates the secondary concern as an inter-type declaration
Replace Implements with Declare Parents	Classes implement an interface related to a secondary concern. Class code implementing the interface is used only when the secondary concern is included in the system build	Replace the implements in the class with a equivalent declare parents in the aspect

Some of the refactorings from Table II serve to remove the *Aspect Laziness* smell [4], i.e. situations when an aspect statically introduces state and behaviour to a set of classes when a more dynamic and unpluggable composition is desirable. This smell can be found in systems that require such flexibility of composition and yet resort to inter-type declarations, which performs the compositions statically. Those refactorings serve to replace this static mapping with a similar mapping programmatically supported by the aspect. Figure 2 illustrates the structural effects performed by those refactorings. The refactorings responsible for replacing the inter-type declarations with a mapping of additional state and behaviour are *Replace Inter-type Field with Aspect Map* and *Replace Inter-type Method with Aspect Method*. Their effect is illustrated in Figure 3. For more details, see [4,7].

Finally, we deal with duplication between multiple aspects by factoring out commonalities to a (in this case reusable) super-aspect, by using the refactorings from Table III. Figure 3 illustrates the structural effects of this group of refactorings. Note that Figure 3 mentions all refactorings from this group, not just the ones used in the example of this paper.

From a certain point, the described process follows two alternative paths, both ending with the AOP design described in Section 4.4. The first path is performed solely in terms of the original code, and the second path adds to the system, at a certain point, the abstract base aspect from [13] (ObserverProtocol). Sections 5.1–5.3 describe the first path in detail, comprising the three phases

Table II. Refactorings to restructure the internals of aspects obtained through extraction processes.

Name	Typical situation	Recommended action
Extend Marker Interface with Signature	An inner interface models a role used within the aspect. You would like the aspect to call a method specific to a type that implements the interface but that is not declared by it	Add an inter-type abstract declaration of the case-specific method signature to the interface
Generalise Target Type with Marker Interface	An aspect refers to case-specific concrete types, preventing it from being reusable	Replace the references to specific types with a marker interface and make the specific types implement the marker interface
Introduce Aspect Protection	You would like an inter-type member to be visible within the declaring aspect and all its sub-aspects, but not outside the aspect inheritance chain	Declare the inter-type member as public and place a declare error preventing its use outside the aspect inheritance chain
Replace Inter-type Field with Aspect Map	An aspect statically introduces additional state to a set of classes, when a more dynamic or flexible link between state and targets would be desirable.	Replace the inter-type declarations with a structure owned by the aspect that performs a map between the target objects and the additional state
Replace Inter-type Method with Aspect Method	An aspect introduces additional methods to a class or interface, when a more dynamic and flexible composition would be desirable	Replace the inter-type method with an aspect method that gets the target object as an extra parameter
Tidy Up Internal Aspect Structure	The internal structure of an aspect resulting from the extraction of a crosscutting concern is sub-optimal	Tidy up the internal structure of the aspect by removing duplicated inter-type declarations and dependencies on case-specific target types

Table III. Refactorings to improve the generalisation of aspects.

Name	Typical situation	Recommended action
Extract Super-aspect	Two or more aspects contain similar code and functionality	Move the common features to a super-aspect
Pull Up Marker Interface	All sub-aspects use a marker interface to model the same role	Move the marker interfaces to the super-aspect
Pull Up Pointcut	All sub-aspects declare identical pointcuts	Move the pointcuts to the super-aspect
Push Down Advice	A piece of advice is used by only some sub-aspects, or each sub-aspect requires different advice code	Move the advice to the sub-aspects that use it

of the refactoring strategy, each relating to a composite refactoring [4] prescribing the use of other refactorings:

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 Super-aspect*: factors out common code from the aspects to an abstract super-aspect.

The above three refactorings are *composite* refactorings: they are more general than most and their purpose is to set the background for the use of other refactorings.

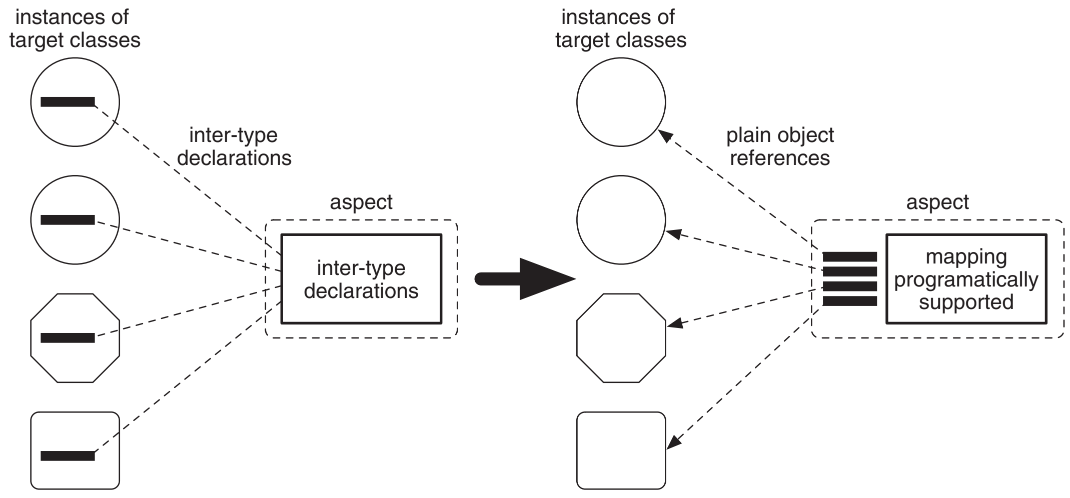
instances of target classes
inter-type declarations
aspect
inter-type declarations
instances of target classes
plain object references
aspect
mapping programatically supported

Figure 2. Illustration of the structural effect obtained by the removal of the aspect laziness smell.

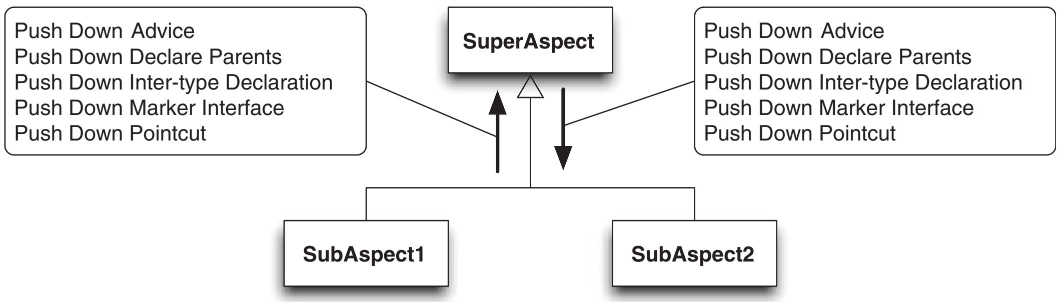
Push Down Advice
Push Down Declare Parents
Push Down Inter-type Declaration
Push Down Marker Interface
Push Down Pointcut
SuperAspect
Push Down Advice
Push Down Declare Parents
Push Down Inter-type Declaration
Push Down Marker Interface
Push Down Pointcut
SubAspect1
SubAspect2

Figure 3. Structural effects performed by the refactorings from Table III.

The second path is described in Section 6. It diverges from the first path from the beginning of the second phase (tidying up). In it, ObserverProtocol is added to the system and therefore *Extract Super-aspect* (third phase of the first path) is not needed.

It is important to note that the refactoring processes described in this paper correspond to just two of many possible paths. Though the finishing point of the various paths should result in an implementation of the intended design in all cases, it is possible to achieve it through multiple different paths, since there are several possible alternatives to choose from after each step.

Throughout the description of the process, code listings and code fragments are used to illustrate relevant details. In many occasions, changes from the previous illustration are highlighted in bold, following the example from [1]. Likewise, deleted sections of code are signalled in strike-through in some occasions.

5.1. First phase: extracting features

The first phase begins with the extraction of the observing relationship associated with Flower.open. There are three inner classes related to this concern (Listings 7 and 8): Flower.OpenNotifier, Bee.OpenObserver and Hummingbird.OpenObserver (not shown). We apply *Extract Inner Class into Standalone* to inner class Flower.OpenNotifier. This refactoring also entails the prior extraction of method Flower.isOpen, using *Extract Method* [1]. Note that the code that creates instances of inner classes is not affected by the refactoring

```
public class Flower {
  private boolean _isOpen;
  private OpenNotifier oNotify =
     new OpenNotifier();
  ...
  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;
     }
  }
```

```
public class Flower {
  private boolean _isOpen;
  private OpenNotifier oNotify =
     new OpenNotifier();
  ...
  boolean isOpen() {
     return _isOpen;
  }
  private void setIsOpen(boolean newValue) {
     _isOpen = newValue;
  }
  ...
```

➡

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

Next, we would like to do the same with classes Bee.OpenObserver and Hummingbird.Open Observer. However, each class contains an action (print a message to the console) that belongs to the primary functionality of its enclosing class. We therefore first use *Extract Method* [1] on both classes, giving rise to two methods breakfastTime. Both inner classes have the same name and are almost identical, so it is feasible to extract them into a single stand-alone class. However, the inner classes hold a field referring to their enclosing classes, which are of different types. For this reason, we use *Extract Interface* [1] on the enclosing classes so that the new stand-alone class can refer to the former enclosing type through the extracted interface. Operations declared by interfaces must be public and for this reason we make both breakfastTime methods public

```
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!");
   }
 }
```

➡

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

```
public class Bee  implements BreakfastTaker {

  //...
  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();
     }
  }
```

Next, we apply *Extract Inner Class into Standalone* and use the new interface as the type of the 'former enclosing object'. 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 visibility 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 to the aspect all code using field oNotify, so it is now possible to make the field private again, but this time relative to the aspect. Code that initializes the moved fields is moved along with them. The aspect now has the following contents:

```
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 is now 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 the concern. We apply *Move Field from Class to Inter-type* to Bee.openObsrv.

Fields are usually private to their owner classes. For this reason, *Extract Feature into Aspect* recommends that fields be moved before methods that use those fields. That entails relaxing the visibility of the fields, as they are temporarily referred by code from multiple modules. In this case, field visibility is relaxed from private to package protected. The documentation of *Move Field from Class to Inter-type* recommends that a 'scouter' declare warning be temporarily added to the aspect, to ensure all points referring to the fields are located:

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

The declare warning signals a use of the field outside the aspect, in method Bee.openObserver. In principle, a method that uses a field related to a concern is likely to belong to that concern. That is indeed the case with Bee.openObserver and therefore we apply *Move Method from Class to Inter-type* to it. The warnings are gone, so the declare warning is removed and the visibility of field openObsrv is changed back to private (to the aspect). Similar refactorings are applied to Hummingbird. Both observers are now devoid of any code related to the first observing relationship, except for the implements clause referring to BreakfastTaker:

```
public class Bee implements BreakfastTaker {
    private String name;
    ~~private OpenObserver openObsrv = new OpenObserver(this);~~
    private CloseObserver closeObsrv = new CloseObserver();

    public Bee(String nm) {
        name = nm;
    }
    public void breakfastTime() {
        System.out.println("Bee " + name + "'s breakfast time!");
    }
    ~~private class OpenObserver implements Observer {~~
  ~~      //...~~
  ~~  }~~
   // 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;
     }
  }
```

At this point, aspect ObservingOpen contains the following code:

```
public aspect ObservingOpen {
  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;
  }
  public Observer Hummingbird.openObserver() {
    return openObsrv;
  }
  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;
  }
  declare error: get(OpenObserver openObsrv) && !within(ObservingOpen):
    "field openObsrv accessed outside aspect.";
}
```

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 warrants further refactorings (see Section 5.3). The second extraction comprises the following steps:

- 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 visibility is temporarily relaxed from private to package protected. This refactoring entails creating a declare warning exposing three points in Flower that still use 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 exposed by the declare warning, so the declare warning is removed and the field Flower.cNotify is made private (to the aspect).

From this point on, Flower is clean of any code related to observing relationships. Next, we deal with the remaining code in the observer participants, Bee and Hummingbird. The first thing to do 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 stand-alone class. This entails: (1) applying *Extract Method* [1] to create method bedtimeSleep in each of them; (2) use *Extract Interface* [1] to extract BedtimeSleeper, just as in the actions that yielded method breakfastTime and interface BreakfastTaker:

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

Now, we can use *Extract Inner Class into Standalone* to both CloseObserver inner classes to yield the common stand-alone class CloseObserver:

```
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.

Some 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. We now use *Encapsulate Implements with Declare Parents* to both Bee and Hummingbird, so that all participants become completely free of any code related to the extracted concerns.

The refactorings performed until now cleaned the participant's code but also created several stand-alone classes and interfaces that provide little functionality and are used only by code placed within the aspects. We therefore inline them so that all codes related to observing relationships are encapsulated in their respective aspects. The former inner classes depend on the interfaces and therefore it is convenient to inline the classes before the interfaces. We apply *Inline Class within Aspect* to classes OpenObserver, CloseObserver, OpenNotifier and CloseNotifier. Next, we use *Inline Interface within Aspect* on interfaces BreakfastTaker and BedtimeSleeper. The code related to both concerns is now completely modularized within their respective aspects (Listing 11).

5.2. Second phase: tidying up the extracted aspects

At this point, the internal structure of the extracted aspects Listing 11) is confusing and needlessly complex. It contains much duplication and several inner classes and interfaces for whose existence there is no longer a compelling reason. In addition, both aspects betray the *Aspect Laziness* smell [4]: they statically attach additional state and behaviour to participant classes, while in this case a dynamic and unpluggable composition is advantageous. With the modularization attained through AOP, the scattering effect is just another code smell that can be removed with refactorings [4].

The aim of the refactoring phase described in this section is to improve the internal structure of the aspects, by removing internal duplication and *Aspect Laziness*, thus bringing the current structure to one closer to current notions of good AO design [13]. These tidying up transformations are prescribed by *Tidy Up Internal Aspect Structure*, to be used to each aspect in turn. This also makes the internal structure of the aspect more amenable to the subsequent use of *Extract Super-aspect*, which removes duplication across multiple aspects. We start by refactoring ObservingOpen. As soon as the process is completed, a similar process is carried out on ObservingClose. We first use *Generalise 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

```
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 11. Aspect ObservingOpen just after the extraction of all code related to observing operation *open*.

and Observer to represent the 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 being called Observer. We therefore remove the import statement referring to java.util.Observer and make all references use the full compound name

```
 import java.util.Observable;             ~~import java.util.Observable;~~
 import java.util.Observer;               ~~import java.util.Observer;~~

public aspect ObservingOpen {              public aspect ObservingOpen {
  //...                                ➡     //...
  static class OpenNotifier                 static class OpenNotifier
       extends Observable {                      extends  java.util.Observable {
    //...                                      //...
  }                                           }
```

We first apply *Generalise Target Type with Marker Interface* to type Flower. We do this by replacing all references to Flower with marker interface Subject, including references found within the various OpenNotifier inner classes. This transformation gives rise to a compiler error: the Subject interface does not declare operation isOpen. As a stopgap, we use *Extend Marker Interface with Signature* on Subject to add the signature of isOpen to Subject. This entails changing the visibility of method Flower.isOpen from package protected to public. Next, we apply *Generalise Target Type with Marker Interface* to Bee and Hummingbird to eliminate all references to the case-specific interface BreakfastTaker, which are replaced with references to marker interface Observer. Next, interface BreakfastTaker is removed

```
public aspect ObservingOpen {              public aspect ObservingOpen {
   //...                                      //...
   private interface BreakfastTaker {       ~~private interface BreakfastTaker {~~
      public void breakfastTime();            ~~public void breakfastTime();~~
   }                                          ~~}~~
   //...                                        //...
  static class OpenObserver                   static class OpenObserver
      implements java.util.Observer {  ➡
                                                  implements java.util.Observer {
    private BreakfastTaker _enclosing;          private  Observer _enclosing;
    public                                      public
    OpenObserver(BreakfastTaker enclosing) {    OpenObserver( Observer enclosing) {
       _enclosing = enclosing;                     _enclosing = enclosing;
    }                                           }
```

Once more we use *Extend Marker Interface with Signature*, to add the case-specific signature of method breakfastTime to Observer. This step also eliminates duplication in method openObserver, which is introduced twice (to Bee and Hummingbird). The aspect now refers to the concrete participants only in the declare parents.

Now that some glaring duplication is removed, the next step is to remove *Aspect Laziness*, by replacing the inter-type state and behaviour with equivalent functionality that is dynamically composable. This entails adding aspect methods akin to the inter-type methods to be replaced and then replace the calls to the original inter-type methods in the client code (located within test code from [11], not shown) with calls to the new aspect methods.

What follows is to some extent an elaborated instance of *Replace Inter-type Field with Aspect Map*. The difference is that in the present case we deal with inner classes rather than inter-type fields. As prescribed in the description of *Replace Inter-type Field with Aspect Map*, we use *Replace Inter-type Method with Aspect Method* as a follow-up (the strategy is first to deal with fields and next with the methods that use those fields). The aim of this sequence of refactorings is to perform the transformation sketched in Figure 3. They add a mapping structure to the aspect, along with the associated logic. Note that the new logic includes one operation (clearObservers) that is not present in the abstract aspect presented in [13]

```
public aspect ObservingOpen {
   private interface Subject {}
   private interface Observer {}
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;

    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();
    }
```

Next, *Replace Inter-type Method with Aspect Method* is used, which entails adding aspect method notifyObservers. The method provides the same functionality as OpenNotifier.notifyObservers, using a boolean field (alreadyOpen) newly added to Subject, to be used for the same purposes as field alreadyOpen of inner class OpenNotifier:

```
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 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();
         }
      }
   }
```

Also as prescribed in *Replace Inter-type Method with Aspect Method*, we add a declare warning to expose all places where the logic to be replaced is located. The declare warning targets method Subject.opening, the accessor method for the instance of inner class OpenNotifier

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

Compiling after the declare warning is added exposes six warnings, all pointing to client (unit test[††]) code. We replace the original calls with calls to aspect logic. One example is as shown:

```
f.opening().addObserver(ha.openObserver());   ➡   ObservingOpen.aspectOf().addObserver
                                                   (f, ha);
```

Tidying up the second aspect: Improving the internal structure of ObservingClose requires essentially the same steps as with ObservingOpen, i.e. another instance of *Tidy Up Internal Aspect Structure*. The steps are:

- Removal of imports of java.util.Observable and java.util.Observer. The compound names are used instead.
- Creation of private inner interfaces Observer and Subject as a preliminary step to using *Generalise Target Type with Marker Interface*.
- Applying *Generalise Target with Marker Interface* to Flower: references to Flower are replaced by references to marker interface Subject.
- While using *Generalise Target with Marker Interface*, *Extend Marker Interface with Signature* is used to introduce method isOpen to Subject. References to Bee and Hummingbird are replaced by references to marker interface Observer. Interface BedtimeSleeper is removed (along with a declare parents that served to add a signature). *Extend Marker Interface with Signature* is used once more to extend type Observer with the signature of bedtimeSleep.
- Use of *Replace Inter-type Field with Aspect Map*, followed by *Replace Inter-type Method with Aspect Method* to add a new implementation to ObservingClose. As prescribed by *Replace Inter-type Method with Aspect Method*, a declare warning is added to expose calls to accessor method CloseNotifier.closing:

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

- Following the locations exposed by the declare warning, the calls to CloseNotifier.closing (in test code, not shown) are replaced. Again, we must reverse the order in which observers are registered. We remove the declare warning, compile and the test passes.

[††]In the course of the transformations described, an assertion within test code fails at a given point, due to two implementations traversing in opposite orders the list of observers to be notified. The order of notification of observers is not relevant but the particular way with which Eckel's original test was adapted needlessly hard coded the original order of notifications. For this reason, the order with which observers are registered in the test is inverted.

5.3. Third phase: factoring out common code to a super-aspect

Though the refactored aspects are better formed, when considered together they betray much duplication across aspects, namely in the marker interfaces, field subject2ObserversMap and associated logic. We eliminate the duplication by using *Extract Super-aspect* to factor out common code to a super-aspect. This entails performing the following steps:

- Create blank abstract aspect ObservingRelationships.
- Aspects ObservingOpen and ObservingClose are made to extend ObservingRelationships.
- Use *Pull Up Marker Interface* on inner interfaces Subject and Observer of both aspects, to move them to ObservingRelationships. Their visibility is relaxed from private to protected.
- Use *Pull Up Field* [1] on the field subject2ObserversMap of both aspects.
- Use *Pull Up Method* [1] on methods getObservers, addObserver, removeObserver and clearObservers of both aspects.

Method notifyObservers is another candidate to being pulled up, but it depends on too many case-specific members. For this reason, we merely extract to the super-aspect its abstract declaration. Pointcuts flowerOpen and flowerClose are also case specific and for this reason we refrain from adding more abstract declarations to the super-aspect. This decision illustrates one of the advantages of refactoring: as the code can be changed in the future, design decisions do not have to be made upfront. Developers have the option to change their minds at a later phase, and refactor. The extracted super-aspect (very similar to ObserverProtocol from [13]; Listing 9) is shown in Listing 12 and ObservingOpen is shown in Listing 13.

```
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 12. Stable form of the ObservingRelationships abstract aspect.

```
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();
      }
    }
  }
  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 13. ObservingOpen after factoring out duplicated code to the super-aspect.

6. ALTERNATIVE REFACTORING PATH

The previous section shows how to use refactoring to gradually transform a Java implementation of Observer into an implementation in AspectJ. However, we also have the option of reusing the ObserverProtocol aspect from [13]. It is equally feasible to take advantage of this ready-made aspect while refactoring the case-specific aspects of the example. In this section, we describe such an alternative refactoring path.

The alternative path starts diverging from the one described in the previous section after the completion of the extraction process (Section 5.1). It almost exclusively involves adding just the case-specific parts, as ObserverProtocol already provides the most of the generally applicable parts. The existing structure must be tidied up as previously, but because ObserverProtocol is included, the alternative path involves adding fewer amounts of new code compared to the previous path.

Again, we tackle one aspect at a time, starting with the use of *Tidy Up Internal Aspect Structure* on ObservingOpen. This refactoring prescribes the use of *Generalise Target Type with Marker Interface*, which requires adding marker interfaces Subject and Observer representing the participant roles in the pattern. This time we do not need to add them as they are declared in ObserverProtocol. Instead, we make ObservingOpen inherit from ObserverProtocol. This in turn gives rise to several issues. One is the conflict caused by two member types being named Observer, which we again solve by removing the import to java.util.Observer and making all references to the interface use the full compound name. Another issue is the need to provide definitions corresponding to the abstract declarations made by ObserverProtocol—pointcut subjectChange and method updateObserver. At this stage, we start with a blank definition of updateObserver and subjectChange is added with the expression that captures the interesting events:

```
   protected pointcut subjectChange(Subject subject):
      execution(void open()) && this(subject);
   protected void updateObserver(Subject s, Observer o) { }
```

In this intermediate stage, the internal structure of aspect ObservingOpen is rather confused, particularly because it contains two alternative implementations of the same logic. Fortunately, modularization places us in a good position to tidy up the internal structure. The first step is to apply *Generalise Target Type with Marker Interface*, which entails assigning the roles to the participants by means of a declare parents:

```
public aspect ObservingOpen extends ObserverProtocol {
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;
```

Next, we use *Generalise Target Type with Marker Interface*, starting with Flower. We replace all references to Flower with Subject, and again we use *Extend Marker Interface with Signature* to extend Subject with the isOpen signature, which in turn compels us again to relax the visibility of method Flower.isOpen from package protected to public. Next, we replace references to Bee and Hummingbird with references to marker interface Observer, which again enables us to replace references to inner interface BreakfastTaker with references to Observer, and remove BreakfastTaker, as well as the declare parents clause that targeted it. We also use *Extend Marker Interface with Signature* again, this time to extend Observer with signature breakfastTime.

```
public                                        ~~public~~
OpenObserver(BreakfastTaker enclosing) {      ~~OpenObserver(BreakfastTaker enclosing) {~~
   _enclosing = enclosing;                    ~~   _enclosing = enclosing;~~
}                                             ~~}~~

                                              public abstract
                                        ➡    void Observer.breakfastTime( );
                                              //...
static class OpenObserver                     static class OpenObserver
      implements java.util.Observer {               implements java.util.Observer {
   private BreakfastTaker _enclosing;            private Observer _enclosing;
   public                                        public
   OpenObserver(BreakfastTaker enclosing) {      OpenObserver(Observer enclosing) {
      _enclosing = enclosing;                       _enclosing = enclosing;
   }                                             }
```

As before, the replacements of Bee and Hummingbird with Observer eliminate duplication in the introductions of field openObsrv and method openObserver. Note that the code passing the self-variable this to the constructor of OpenObserver (which now refers to an instance of Observer) only compiles because all references to BreakfastTaker were first replaced with Observer

```
private OpenObserver Hummingbird.openObsrv = new OpenObserver(this);
private OpenObserver Bee.openObsrv = new OpenObserver(this);
//...
public java.util.Observer Bee.openObserver() {
   return openObsrv;
}
public java.util.Observer Hummingbird.openObserver() {
   return openObsrv;
}
```

⇓

```
~~private OpenObserver Hummingbird.openObsrv = new OpenObserver(this);~~
private OpenObserver Observer.openObsrv = new OpenObserver(this);
//...
~~public java.util.Observer Bee.openObserver() {~~
~~   return openObsrv;~~
~~}~~
public java.util.Observer Observer.openObserver() {
   return openObsrv;
}
```

This completes the application of *Generalise Target Type with Marker Interface* to types Hummingbird, Bee and Flower. From this point on, participants are referred only in the role-assigning declare parents clause. In the process, pointcut flowerOpen is removed as well, as it is identical to subjectChange. Further improvements on the internal structure of ObservingOpen require the replacement of the current implementation with the one defined in ObserverProtocol. Only then it is possible to remove the inner classes and the dependence on the Observable/Observer protocol from java.util.

There are a few hurdles. 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*, to account for the two different operations of Flower. 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 occurrence of a sequence). Only if the test succeeds are the subject's registered observers notified. This test relies on boolean field alreadyOpen of inner class OpenNotifier. The field should be moved to Flower, as an inter-type declaration, but the point where the test is made lies within the abstract base aspect ObserverProtocol, in a piece of advice acting on the joinpoints captured by pointcut subjectChange:

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

Therefore, ObserverProtocol must incur invasive changes to be used in this example. A sub-aspect cannot override advice inherited from the super-aspect and therefore the advice must be pushed down from ObserverProtocol to ObservingOpen, through *Push Down Advice*. In addition, functionality is missing in ObserverProtocol: the ability to clear all observers subscribing to a given subject. We therefore add the method clearObservers (identical to the one added during the tidying up phase described in Section 5.2) to ObserverProtocol.

ObservingOpen now has two pieces of after advice acting on this pointcut, related to the old and new implementations, respectively. Two advice of the same kind acting on the same pointcut within a single aspect clearly seems bad style—in the normal case, the two advice blocks should merge into a single advice, perhaps with calls to suitably named auxiliary methods. However, the advice related to the original implementation is about to be removed. We now add the missing logic to ObservingOpen. It is based on the alreadyOpen field introduced to Subject. This involves adapting the advice pulled down from ObserverProtocol according to the rules of this case:

```
public aspect ObservingOpen extends ObserverProtocol {
  protected pointcut subjectChange(Subject subject):
    execution(void Subject+.open()) && this(subject);
  after(Subject s): subjectChange(s) {
    Flower f = (Flower)s;
    if(f.isOpen() && !f.alreadyOpen) {
       Iterator iter = getObservers(s).iterator();
       while ( iter.hasNext() ) {
          updateObserver(s, ((Observer)iter.next()));
       }
   }
}
```

We add the advice acting on the pointcut that captures the execution of method Flower.close:

```
public aspect ObservingOpen extends ObserverProtocol {
  //...
  pointcut flowerClose(Subject flower): execution(void close()) && this(flower);
  after(Subject flower): flowerClose(flower) {
    flower.oNotify.close();
  }
   after(Subject subject): flowerClose(subject) {
       if (subject instanceof Flower) {
         ((Flower)subject).alreadyOpen = false;
       }
   }
```

Finally, we provide aspect method updateObserver with the logic suitable for the new implementation:

```
public aspect ObservingOpen extends ObserverProtocol {
 //...
 protected void updateObserver(Subject s, Observer o) {
     o.breakfastTime();
     ((Flower)s).alreadyOpen = true;
 }
```

We now remove the original implementation. We could consider one last tidying up to do, related to the breakfastTime signature that is still extending interface Observer. Though we could replace

```
public aspect ObservingOpen extends ObserverProtocol {
   declare parents: Flower implements Subject;
   declare parents: (Bee || Hummingbird) implements Observer;

   protected pointcut subjectChange(Subject subject):
      execution(void Subject+.open()) && this(subject);
   after(Subject s): subjectChange(s) {
      Flower f = (Flower)s;
      if(f.isOpen() && !f.alreadyOpen) {
         Iterator iter = getObservers(s).iterator();
         while ( iter.hasNext() ) {
            updateObserver(s, ((Observer)iter.next()));
         }
      }
   }
   pointcut flowerClose(Subject flower): execution(void close()) && this(flower);
   after(Subject subject): flowerClose(subject) {
      if (subject instanceof Flower) {
         ((Flower)subject).alreadyOpen = false;
      }
   }
   protected void updateObserver(Subject s, Observer o) {
      o.breakfastTime();
      ((Flower)s).alreadyOpen = true;
   }
   public abstract void Observer.breakfastTime();
   private boolean Subject.alreadyOpen = false;
}
```

Listing 14. Final version of ObservingOpen obtained from the alternative path.

it with type conversions in the points where breakfastTime is used, it is proving rather convenient because it is affecting two different and unrelated types, thus avoiding duplication. We choose to leave it there for now. Listing 14 shows the final version of ObservingOpen. We now perform a similar sequence of steps to the second aspect, yielding similar results. This time there is no need to make more methods public.

7. DISCUSSION

We base our work on the hypothesis that evolving a complex collection of scattered code fragments in systems of realistic dimensions is a costly and risky process and that the modularization of such collections, made possible by AOP, bring benefits to comprehensibility and evolution [33]. We assume that the larger is the system, the larger is the expected benefit to comprehensibility and ease of evolution that modularization can bring.

Hannemann [13] analyses the GoF patterns and their AspectJ implementations according to the roles defined by the pattern. The pattern roles are classified as *superimposed* when the object has a distinct, primary role besides the one assigned by the pattern. Otherwise, pattern roles are classified are *defining*, i.e. the object exists only to play the role in the pattern. Such roles do not lend themselves for a separation of roles because the participant plays only that role. The authors acknowledge that role classification into superimposed and defining is not clear-cut in some cases. Classes with superimposed roles betray the *Double Personality* smell as we define it in [4,9].

Conceptually, superimposed roles can be plugged and unplugged from existing objects. However, OO languages are often unable to do that, because they do not modularize superimposed roles such as those defined by *Observer*. Instead, code related to the roles is scattered throughout participant classes. In terms of reusability and ease of evolution, it seems reasonable to assume that patterns defining superimposed roles will derive significant benefits from AOP. Among the GoF patterns, the benefits brought by *Observer* are particularly noticeable, as it is the only pattern that includes *two* superimposed roles, in addition to not including any defining roles. Consequently, the AspectJ implementation of *Observer* modularizes a whole *collaboration* of objects. That is one reason why in this case we observe a significant improvement over traditional OO.

Experience gained from using *Extract Feature into Aspect* shows that extractions of class members based on inter-type declarations do not change the original design, they merely isolate its various elements in a single module at the source code level. This makes the internal structure of the extracted aspect awkward to deal with and may betray the *Aspect Laziness* smell [4]. Such modules can benefit from changes to its internal structure, if not a downright replacement of the original internal design and implementation. That provides the motivation for using *Tidy Up Internal Aspect Structure* and modularization is a prerequisite for applying it.

The refactoring process described in this paper also suggests that it is hard to derive reusable modules, even with AOP. Though the abstract aspect from [13] is potentially reusable, it had to undergo invasive changes in order to adapt it to the simple Java example by Eckel [11].

As regards the obliviousness property [32], participant classes do become oblivious to roles defined by the pattern after the system is refactored to AspectJ. However, it is still necessary for some part of the system to remain aware of the fact that participant classes play the additional roles, in order to set relationships and perform configurations. In Eckel's example, that role is played by test code (not shown in the paper); in the example by Hannemann and Kiczales [13], it is played by a main method. Those parts are responsible for calling aspect methods to register and unregister observers to a given subject. Conceptually, such parts reside at a different level than participant classes and it seems reasonable to expect them not to be oblivious to the aspects.

It is important to note that the definition of obliviousness proposed in [32] applies not only to a set of classes but also to the programmers that develop the classes. We do not abide by that more demanding view of obliviousness. Although classes do not betray code that is dependent on aspects throughout the process, the code base must be refactored in order for it to expose the joinpoint leverage that aspects need. One example is the isOpen method, which was extracted from the class playing the role of Subject (see Section 5.1). Our experiments suggest that although class obliviousness is desirable and often achievable, programmer obliviousness raises important issues related to the development process and does not seem to be feasible in practice. We give the name *aspect-friendly* [3] to a code base that is refactored to provide the necessary leverage to aspects but remains code oblivious to any specific aspect code (i.e. a system build can be performed without including the aspects in it).

The experience gained from deriving the refactoring process described in this paper, as well as other experiments [9], suggests that manual extraction of CCCs in systems of realistic dimensions involves a significant amount of work. For this reason, we think it is highly desirable that automated support [34] be provided in tools and developing environments. Future work includes testing the refactorings illustrated in the paper in larger and more complex systems.

8. RELATED WORK

The refactoring process described in this paper comprises a first use and illustrating example of a collection of novel refactorings for AOP code that we describe in [3,4]. In [4], we propose a few novel code smells, including *Double Personality* and *Aspect Laziness*. Several other authors report on the successful reuse of those refactorings. van Deursen *et al.* [36] describe the analysis of the JHotDraw framework[‡‡] and the extraction of several CCCs from its code base. They report using *Encapsulate Implements with Declare Parents* and *Move Method from Class to Inter-typ*e. Fuentes *et al.* [37] report on refactoring the Java code base of an ambient intelligence application and mention benefits derived from our refactorings [4,9]. Likewise, Kulesza *et al.* [38] report on refactoring the JUnit framework[§§] mainly on the basis of those refactorings.

Hannemann [13] presents implementations of the 23 GoF patterns [10] in both Java and AspectJ. They were able to modularize the implementation of 17 of the patterns, 12 of which had part of the implementations abstracted into reusable aspect modules. In a few cases multiple instances of the aspect can be transparently composed into a system. *Observer* is one of the patterns, whose design is used as the target of the refactoring processes described in this paper. The authors present an analysis of the two sets of implementations and conclude that improvements of the AspectJ implementations over the Java ones are directly correlated to the presence of crosscutting structure in the patterns. The crosscutting effect arises in patterns that superimpose additional (secondary) roles on participant classes and whose implementation code cuts across participant classes.

Hannemann *et al.* [34] and again Hannemann [35] propose that refactoring support for AOP be divided into three categories: *aspect-aware OO refactorings* (a concept previously proposed by Hanenberg *et al.* [5]), *aspect-oriented refactorings* (i.e. refactorings that specifically target AO constructs, such as those used here) and *refactorings of* CCCs, i.e. refactorings in which the scattered elements comprising a target CCC and their individual transformations are considered together, instead of handling each element separately. It does not seem possible to carry out, as a single operation, the latter category of refactorings without the support of a tool and the focus of [34] is to present one such tool. However, a *sequence* of AO refactorings, such as the process described in this paper, achieves a similar effect as a refactoring of CCCs as understood by Hannemann.

Like us, Hannemann *et al.* [34] use *Observer* as a basis for an illustrating example of a refactoring process that results in the modularization of an implementation of *Observer*. They provide an outline of the steps to be carried out, but it is significantly less detailed than the one we present in this paper. The outcome of their illustrating refactoring is the AspectJ design for *Observer* presented in [13], which we also use in this paper. As it would be expected, there are similarities between the transformations presented in the outline and the ones we describe here. However, more details than those provided in [34] are required for a thorough comparison and analysis.

Several other authors use the GoF patterns, again with an emphasis on *Observer*, to illustrate the relative advantages of AOP [4,39–41]. The comparison presented in [40] is interesting in that the comparison made is not between OO and AO implementations of *Observer*, but between the AspectJ implementation proposed in [13] and a different AO implementation coded in Caesar.

[‡‡] www.jhotdraw.org.
[§§] www.junit.org.

The comparison stresses the *disadvantages* of the AspectJ implementation relative to the Caesar implementation as regards internal modularity, polymorphism and compositional capabilities.

Iwamoto and Zhao [6] announce their intention to build a catalogue of AOP refactorings. They present a catalogue of 24 refactorings, but the information provided about them is limited to the names of the refactorings. In this paper, we provide more detail about the AO refactorings used throughout the process described in this paper, even if we refer the full documentation to [3,4,9].

Hanenberg *et al.* [5] also propose three AOP refactorings—*Extract Advice*, *Extract Introduction* and *Separate Pointcut*. Their proposed *Extract Advice* corresponds to our *Extract Fragment into Advice* and their proposed *Extract Introduction* corresponds to our *Move Field from Class to Inter-type* and *Move Method from Class to Inter-type*. *Separate Pointcut* relates to evolution of pointcuts and has no correspondence in our collection. The latter refactoring concerns good style for pointcuts, expressing the notions that pointcuts should in most situations be named rather than anonymous, and be as decomposed as possible to enhance their reusability. The latter notion is also argued by Lagaisse and Joosen [42]. *Separate Pointcut* is not needed in the process described in this paper because anonymous pointcuts are avoided. In addition, the declaration and use of pointcuts is in part constrained by the existing interface of ObserverProtocol.

Laddad [7] presents a collection of refactorings tailored to J2EE applications, covering a different area of the AO refactoring space than the refactorings used in the process described in this paper. The refactorings vary widely in both level and scope of applicability, including generally applicable refactorings like *Extract Interface Implementation*, *Extract Method Calls* and *Replace Override with Advice*, but also concern-specific refactorings such as *Extract Concurrency Control* and *Extract Contract Enforcemement*. Some refactorings target specific AO design patterns [28], e.g. *Extract Worker Object Creation* and *Replace Argument Trickle by Wormhole* and *Extract Exception Handling*. Some of the refactorings are presented with only a mention to the name and a brief motivating paragraph. van Deursen *et al.* [36] report on using *Extract Method Calls*. Kellens and Gybels [43] provide a description of *Extract Method Calls* that is more detailed and analyse the refactoring in the light of its automated application.

van Deursen *et al.* [36] propose a few new refactorings based on their work on JHotDraw, though without providing much detail. These include *Move Role to Aspect*, which entails moving code related to a secondary role to an aspect. This is actually a refactoring that removes *Double Personality*. The refactorings we propose in [4] are more low level but can achieve similar results through a suitable sequence. van Deursen *et al.* also propose *Move Observer to Aspect*, a more high level compound refactoring that moves an entire implementation of Observer to an aspect. In addition, the authors propose *Override Method with Advice for Overlapping Roles*, which applies to methods that perform multiple features relating to multiple roles. The refactoring extracts a method definition to the Java interface to which the method belongs. An aspect uses an around advice to override the default behaviour provided by the extracted method in the situations where it is suitable. Finally, the authors propose *Advise Method Overrides*, a refactoring that extracts to an aspect the idiom comprising duplicated statements common to the (start or end of) all method overrides of a given (superclass) method.

There are a number of works describing experiments in refactoring existing OO code bases, with various degrees of detail provided. In [3], we report on our experiments in extracting a concern from a Java framework for workflow applications. Tonella and Ceccato [44] report on the results obtained in extracting the implementation of interfaces (approached as symptoms of latent aspects) from the source code of some of the packages in the standard Java library. Bruntink *et al.* [45]

report on the refactoring of specific concern—parameter checking—to separate modules coded in a domain-specific language. Colyer and Clement [46] describe an experiment in refactoring a middleware product line with tens of thousand classes, many millions of lines of code and hundreds of developers. Zhang and Jacobsen [47] describe the refactoring they performed of ORBacus, an industrial strength CORBA implementation.

Cole and Borba [48] propose programming laws from which refactorings for AspectJ can be derived. The authors focus on the use of their laws to derive existing refactorings such as those proposed in [3,4], and describe two case studies in which the laws were tested, comprising the extraction of concurrency control and distribution, respectively. Most of the proposed laws relate to the extraction of CCCs to aspects, and therefore there is some overlap between the refactorings they derive and the extraction refactorings used in Section 5.1. However, their focus is on providing proofs that the transformations are behaviour preserving, while we focus on developing a notion of style for AOP, by increasing its refactoring space. Cole and Borba remark that the extraction procedure for the second case study is generalizable, as the implementation of distribution is commonly used. The authors claim that it is possible to derive a concern-specific *Extract Distribution* refactoring, though details are not provided.

9. CONCLUSION

This paper presents a practical example of a refactoring process that includes the extraction of CCCs to aspects, the subsequent internal restructuring of the extracted aspects and the factoring out of common code to super-aspects. The process described serves as an introduction to the collection of refactorings documented in [4,9], playing a similar role to chapter 1 of [1]. It also complements the various code examples found in [4]. Though the paper was written to stand on its own, it refers to an eclipse project available as an online supplement, containing over 30 complete code snapshots. After the description of the process, we provide a discussion and a survey of related work.

ACKNOWLEDGEMENTS

This work was supported by FCT (the Portuguese Foundation for Science and Technology) and FEDER (the European Regional Development Fund) under contract SOFTAS (POSC/EIA/60189/2004).

REFERENCES

1. Fowler M, Beck K, Opdyke W, Roberts D. *Refactoring—Improving the Design of Existing Code*. Addison-Wesley: Reading, MA, 1999.
2. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J, Irwin J. Aspect-oriented programming. *Proceedings of the 11th European Conference on Object-oriented Programming*, Jyväskylä, Finland, 1997 (*Lecture Notes in Computer Science*, vol. 1241), Aksit M, Matsuoka S (eds.). Springer: Berlin, Germany, 1997; 220–242.
3. Monteiro MP, Fernandes JM. Object-to-aspect refactorings for feature extraction. Industry track paper at the *3rd International Conference on Aspect-oriented Software Development*, Boston, MA, 2004.
4. Monteiro MP, Fernandes JM. Towards a catalogue of aspect-oriented refactorings. *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, Chicago, IL, 2005. ACM Press: New York, NY, 2005; 111–122. DOI: 10.1145/1052898.1052908.

5. Hanenberg S, Oberschulte C, Unland R. Refactoring of aspect-oriented software. *Fourth Annual International Conference on Object-oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, Thuringia, Germany, 2003. Springer: Berlin, Germany, 2003; 19–35.
6. Iwamoto M, Zhao J. Refactoring aspect-oriented programs. *Fourth AOSD Modeling with UML Workshop at UML 2003*, San Francisco, CA, 2003.
7. Laddad R. Aspect-oriented Refactoring, Parts 1 and 2, The Server Side, 2003. http://www.theserverside.com/tt/articles/ article.tss?l=AspectOrientedRefactoringPart1, http://www.theserverside.com/tt/articles/article.tss?l=AspectOrientedRefactoringPart2 [25 April 2007].
8. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. *Proceedings of the 15th European Conference on Object-oriented Programming*, Budapest, Hungary, 2001 (*Lecture Notes in Computer Science*, vol. 2072), Knudsen JL (ed.). Springer: Berlin, Germany, 2001; 327–335.
9. Monteiro MP. Refactorings to evolve object-oriented systems with aspect-oriented concepts. *PhD Thesis*, Universidade do Minho, Braga, Portugal, 2005.
10. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns, Elements of Reusable Object-oriented Software*. Addison-Wesley: Reading, MA, 1995.
11. Eckel B. *Thinking in Patterns*, revision 0.9. Book in progress, available online. http://www.pythoncriticalmass.com/downloads/TIPatterns-0.9.zip [16 June 2007].
12. Cooper J. *Java Design Patterns: A Tutorial*. Addison-Wesley: Reading, MA, 2000.
13. Hannemann J, Kiczales G. Design pattern implementation in Java and AspectJ. *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, Seattle, WA, 2002. ACM Press: New York, NY, 2002; 161–172. DOI: 10.1145/582419.582436.
14. Eckel B. *Thinking in Java* (4th edn). Prentice-Hall: Englewood Cliffs, NJ, 2006.
15. Miles R. *AspectJ Cookbook*. O'Reilly: Sebastopol, CA, 2005.
16. Tourwé T, Brichau J, Gybels K. On the existence of the AOSD-evolution paradox. *Workshop on Software-engineering Properties of Languages for Aspect Technologies at AOSD 2003*, Boston, MA, 2003.
17. Koppen C, Störzer M. PCDiff: Attacking the fragile pointcut problem. *Proceedings of the Interactive Workshop on Aspects in Software*, Berlin: Germany, 2004.
18. Wloka J. Aspect-aware refactoring tool support. *Proceedings of the Workshop on Linking Aspect Technology and Evolution at AOSD 2005*, Chicago, IL, 2005.
19. Wloka J. Towards tool-supported update of pointcuts in AO refactoring. *Proceedings of the Workshop on Linking Aspect Technology and Evolution Revisited at AOSD 2006*, Bonn, Germany, 2006.
20. Monteiro MP, Fernandes JM. Refactoring a Java code base to AspectJ: An illustrative example. *Proceedings of the IEEE International Conference on Software Maintenance*, Budapest, Hungary, 2005. IEEE Computer Society Press: Los Alamitos, CA, 2005; 17–26. DOI: 10.1109/ICSM.2005.75.
21. Griswold WG. Program restructuring as an aid to software maintenance. *PhD Thesis*, University of Washington, 1991.
22. Opdyke WF. Refactoring object-oriented frameworks. *PhD Thesis*, University of Illinois at Urbana-Champaign, IL, 1992.
23. Opdyke WF, Johnson RE. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. *Proceedings of the Symposium on Object-oriented Programming Emphasizing Practical Applications*, Poughkeepsie, NY, 1990; 145–160.
24. Griswold WG, Notkin D. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology* 1993; **2**(3):228–269. DOI: 10.1145/152388.152389.
25. Wake W. *Refactoring Workbook*. Addison-Wesley: Reading, MA, 2005.
26. Elrad T (moderator) with panelists Aksit M, Kiczales G, Lieberherr K, Ossher H. Discussing aspects of AOP. *Communications of the ACM* 2001; **44**(10):33–38. DOI: 10.1145/383845.383854.
27. Filman RE, Elrad T, Clarke S, Aksit M. (eds.). *Aspect-oriented Software Development*. Addison-Wesley: Reading, MA, 2005.
28. Laddad R. *AspectJ in Action—Practical Aspect-oriented Programming*. Manning: Greenwich, CT, 2003.
29. Colyer A. AspectJ. *Aspect-oriented Software Development*, Filman RE, Elrad T, Clarke S, Aksit M (eds.). Addison-Wesley: Reading, MA, 2005; 123–143.
30. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. Getting started with AspectJ. *Communications of the ACM* 2001; **44**(10):59–65. DOI: 10.1145/383845.383858.
31. Martin M, Livshits B, Lam MS. Finding application errors and security flaws using PQL: A program query language. *Proceedings of the 20th Annual ACM Conference on Object-oriented Programming, Systems, Languages, and Applications*, San Diego, CA, 2005. ACM Press: New York, NY, 2005; 365–383. DOI: 10.1145/1094811.1094840.
32. Filman RE, Friedman DP. Aspect-oriented programming is quantification and obliviousness. *Aspect-oriented Software Development*, Filman RE, Elrad T, Clarke S, Aksit M (eds.). Addison-Wesley: Reading, MA, 2005; 21–35.
33. Garcia A, Sant'Anna C, Figueiredo E, Kulesza U, Lucena C, Staa A. Modularizing design patterns with aspects: A quantitative study. *Transactions on Aspect-oriented Software Development* (*Lecture Notes in Computer Science*, vol. 3880), Rashid A, Aksit M (eds.). Springer: Berlin, Germany, 2006; 36–74. DOI: 10.1007/11687061_2.

34. Hannemann J, Murphy G, Kiczales G. Role-based refactoring of crosscutting concerns. *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, Chicago, IL, 2005. ACM Press: New York, NY, 2005; 135–146. DOI: 10.1145/1052898.1052910.
35. Hannemann J. Aspect-oriented refactoring: Classification and challenges. *LATEr Workshop at AOSD 2006*, Bonn, Germany, March 2006.
36. van Deursen A, Marin M, Moonen L. A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw. *Technical Report SEN-R0507*, Centrum voor Wiskunde en Informatica, 2005.
37. Fuentes L, Jimenez D, Pinto M. Experiences refactoring ambient intelligence applications with aspects. *LATE Workshop at AOSD 2005*, Chicago, IL, 2005.
38. Kulesza U, Sant'Anna C, Lucena C. Refactoring the JUnit framework using aspect-oriented programming. *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming*, *Systems*, *Languages*, *and Applications*, San Diego, CA, 2005. ACM Press: New York, NY, 2005; 136–137. DOI: 10.1145/1094855.1094901.
39. Nordberg II IM. Aspect-oriented dependency management. *Aspect-oriented Software Development*, Filman RE, Elrad T, Clarke S, Aksit M (eds.). Addison-Wesley: Reading, MA, 2005; 557–584.
40. Mezini M, Ostermann K. Untangling crosscutting models with Ceaser. *Aspect-oriented Software Development*, Filman RE, Elrad T, Clarke S, Aksit M (eds.). Addison-Wesley: Reading, MA, 2005; 165–199.
41. Lesiecki N. *Enhance Design Patterns with AspectJ*, Part 2 (*AOP@Work Series at Developerworks*), IBM, 2005. http://www-128.ibm.com/developerworks/java/library/j-aopwork6/index.html [25 April 2007].
42. Lagaisse B, Joosen W. Decomposition into elementary pointcuts: A design principle for improved aspect reusability. *SPLAT*! *Workshop at AOSD 2006*, Bonn, 2006.
43. Kellens A, Gybels K. Issues in performing and automating the 'extract method calls' refactoring. *SPLAT! Workshop at AOSD 2005*, Chicago, IL, 2005.
44. Tonella P, Ceccato M. Migrating interface implementation to aspects. *Proceedings of 20th IEEE International Conference on Software Maintenance*, Chicago, IL. IEEE Computer Society Press: Los Alamitos, CA, 2004; 220–229. DOI: 10.1109/ICSM.2004.1357806.
45. Bruntink M, van Deursen A, Tourwé T. Isolating idiomatic crosscutting concerns. *Proceedings of the 21st IEEE International Conference on Software Maintenance*, Budapest, Hungary, 2005. IEEE Computer Society Press: Los Alamitos, CA, 2005; 322–329. DOI: 10.1109/ICSM.2005.57.
46. Colyer A, Clement A. Large-scale AOSD for middleware. *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, Lancaster, U.K., 2004. ACM Press: New York, NY, 2004; 56–65. DOI: 10.1145/976270.976279.
47. Zhang C, Jacobsen H. Re-factoring middleware systems: A case study. *Proceedings of the International Symposium on Distributed Objects and Applications*, Catania, Italy, 2003 (*Lecture Notes in Computer Science*, vol. 2888), Meersman R, Tari Z, Schmidt DC (eds.). Springer: Berlin, Germany, 2003; 1243–1262.
48. Cole L, Borba P. Deriving refactorings for AspectJ. *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, Chicago, IL, 2005. ACM Press: New York, NY, 2005; 123–134. DOI: 10.1145/1052898.1052909.