CATALOGUE OF REFACTORINGS FOR ASPECTJ

TECHNICAL REPORT UM-DI-GECSD-200401

Miguel Pessoa Monteiro
Ph.D. student

ESCOLA DE ENGENHARIA
Departamento de Informática

UNIVERSIDADE DO MINHO
Campus de Gualtar
4710-057 BRAGA
PORTUGAL

Catalogue of Refactorings for AspectJ

Technical Report UM-DI-GECSD-200401

Miguel Pessoa Monteiro

Ph.D. student
Departamento de Informática, Escola de Engenharia
Universidade do Minho
mmonteiro@di.uminho.pt

Professor adjunto
Escola Superior de Tecnologia de Castelo Branco
Instituto Politécnico de Castelo Branco
mpm@est.ipcb.pt

August 2004

INDEX

1. INTRODUCTION

This report is part of an effort with the aim of finding suitable rules of good style for aspect-oriented programming (AOP). The style rules are expressed by way of a catalogue of refactorings and code smells, following the trend started in the book by Fowler et al [8]. The tool adopted for these efforts is AspectJ, the most mature aspect-oriented programming language presently available. The effort is to be carried out through experiments based on suitable case studies. The aim of the present report is to present the catalogue of refactorings that resulted from our work on two case studies.

Both case studies were undertaken under the assumption that existing Java code bases can be treated as examples of bad-style AspectJ. By that assumption, such code bases can benefit from structural and style improvements carried out through refactoring processes, the same way as an existing bad-style Java code base can be structurally improved, as illustrated in chapter 1 of Fowler's book [8].

The first case study comprised WorkSCo [7], a real application in the area of workflow. Though this experiment yielded some results [14], which are included here, it did not prove to be as rich in yielding insights as initially hoped. The continuing use of WorkSCo as subject of a case study was also compromised by the decision taken by its development team to adopt the AHEAD tool suite [1]. The use of AHEAD meant that Java code was generated, rather than the actually written by the programming team. It is nonsensical to apply refactorings to generated code and therefore our motivation to use WorkSCo as a case study disappeared.

The aforementioned problems and insights led us to select the code presented in [11] as the second case study. This code base (version 1.1) comprises the implementations of the 23 Gang-of-Four design patterns [9] in both Java and AspectJ. The bulk of the material presented in this report stems from our study of that code base.

This report covers the refactorings only. It presents neither code smells [8] nor reports of the validation of the catalogue.

2. THE CATALOGUE

We aimed to present the refactorings in a format that programmers could recognise, to ensure they are immediately applicable. For this reason the refactorings are presented in such a way that they become a natural extension of the one used in Fowler's book [8]. The refactorings were likewise given names in a similar style. In addition, we cross-reference the refactorings, including this report's page number. Whenever a refactoring from [8] is mentioned, we include both the reference and the page number. The presentation structure used in this report comprises the following elements:

- Name of the refactoring.
- Brief mention of a typical situation.
- Brief description of the recommended action.
- Motivation, occasionally complemented with a Preconditions section.
- Mechanics.
- Code Example(s) (except in the simplest refactorings).

In some refactorings we chose to complement the Motivation section with a Preconditions section in order to make more clear the refactoring's scope of applicability.

The purpose of the code examples is to illustrate the use of each refactoring in the proper context. They are not meant to be self-contained and, just as in the examples presented in [8], the resulting code at the end is not necessarily problem-free. We also follow the example of [8] in highlighting the changed code in bold in some situations where it helps to make the changes easier to spot. We hope to complement the code examples presented here with a forthcoming practitioner's report [15].

Some procedure guidelines are generally applicable to all refactoring processes, independently of the transformation being carried out. Techniques such as the following are essential, considering that presently there is no adequate tool support for AOP refactorings.

The first guideline is to ensure the base code exposes all the desirable joinpoints. An effective way to do this is to ensure the code adheres to the style advocated in [8], namely the placement of each relevant concept in its on class, and the use of small methods with meaningful names. Fewer and bigger classes with long methods make it more likely that necessary context will take the form of local method variables rather than object fields, thus preventing the necessary context from being available for capture at the appropriate moment. Long methods make it more likely that AspectJ will not be able to insert the additional behaviour in the right place. Besides making the system easier to understand, using the appropriate style also enriches the system with the desirable joinpoints (e.g. the beginning and end of methods, method parameters, return values, object fields, etc). A second guideline is to ensure the programs are adequately unit tested [3][8] before applying manual code transformations. Good test coverage remains as essential to the refactoring process as before, if no more so.

In [13] Laddad prescribes several guidelines to ensure the refactorings are applied in a safe way. These involve the creation of a first version of a pointcut through the case-by-case enumeration of each interesting joinpoint, followed by its subsequent refactoring to replace it with a semantically more meaningful pointcut – typically based on expressions containing wildcards. To assist in this refactoring, Laddad provides a recipe based on AspectJ's 'declare error' mechanism to verify whether two different pointcuts p1 and p2 capture exactly the same set of joinpoints:

```
declare error:  (!p1() && p2()) || (p1() && !p2()): "Mismatch in join points captured";
```

Prior to extracting crosscutting behaviour from large systems, Laddad also proposes the use of advice deleting the behaviour about to be extracted (through around advice without calls to proceed), providing another opportunity to monitor the effects of the aspect.

We assume the user of the refactorings is using a modern IDE with support for AspectJ (such as eclipse/AJDT), providing services such as structure views, various kinds of searches and the existing support for plain Java refactorings. These services, particularly structure views, are essential for any refactoring process targeting large and complex systems. These services can be complemented by the use of 'declare warning' clauses, which we prescribe at the appropriate points in the mechanics sections of some of the refactorings. All these services play an important role, considering that the code related to the concerns targeted by AOP is, by definition scattered throughout multiple units of modularity.

The mechanics sections of refactorings do not attempt to cover all possible situations that can potentially arise in source code. For instance, they do not cover uses of reflection. Likewise, they do not deal with what we call the *fragile base code problem* i.e. the fact that almost any refactoring can potentially break existing pointcuts. For instance, *Move Method from Class to Inter-type* (19) can break pointcuts using the within() pointcut designator. We believe this problem can only be thoroughly dealt with new kinds of tool support. However, we also believe it is possible to keep this problem under control provided the adequate practices are followed, including programming AspectJ's constructs (particularly pointcuts) with a prudent and appropriate style. Though we warn of a few potential problems, we expect programmers to rely on their knowledge of the code base to check potential trouble spots.

Though the main source of insights comprised implementations of design patterns, all the refactorings presented in this report are generally applicable, with one exception. The collection of refactorings is structured in three broad groups, plus the special case. The groups are (1) refactorings for the extraction of crosscutting concerns from their plain Java code bases to aspects (section 2.1), (2) refactorings relating to the improvement of the internal structure of aspects (section 2.2), and (3) refactorings to deal with generalisation of aspects (section 2.3). The special case (section 2.4) comprises the only case-specific refactoring in this report, dealing with a situation that can arise in legacy code with published interfaces.

2.1. Refactorings for Feature Extraction

The refactorings presented in this section deal with moving the various implementation elements from their original places into aspects. They comprise the starting point of any refactoring process dealing with plain Java code bases. Our experience suggests these will be the most frequently used refactorings. The main refactoring of this group is *Extract Feature into Aspect* (5), covering the general feature extraction algorithm, with the remaining refactorings refining its various steps.

The "open-class" mechanism of inter-type declarations makes it particularly easy to move elements to aspects. From the point of view of client code, there is no difference between a public method declared in its own class or introduced by an aspect. There is no need to scan client code in search for potential problems the move may have caused: we know from the start it didn't cause any. The same ease applies, to varying degrees, to other elements.

The movement of methods and fields are among the most "obvious" refactorings, as well as the extraction of snippets of code to advice. As it would be expected, several proposals for such refactorings have been presented [12][10]. We present our own proposals in this report: *Move Method from Class to Inter-type* (19) to move methods to aspects, *Move Field from Class to Inter-type* (17) to move fields to aspects, and *Extract Fragment into Advice* (9) to move a code snippet to an advice in the aspect.

However, our work revealed that the elements of implementation go beyond just methods and fields. The limitations in object-oriented programming (OOP) sometimes lead programmers to use inner classes to cope with code duplicated in multiple classes. OOP does not enable programmers to modularise the code as much as they would like, but they still can use inner classes to tidy up the internals of each of the involved classes, by providing a better separation between the secondary code from the one related to the primary concern. With AOP we can go further, using *Extract Inner Class to Standalone* (13) to turn the inner class into a standalone class, and next using *Inline Class within Aspect* (15) to place it inside the aspect.

A frequent technique to organise the code and make intentions clear is to use interfaces to represent roles played by classes. With AOP, we can completely modularise the code related with secondary roles, to which those interfaces are associated. Sometimes those interfaces can continue to be useful as maker interfaces within an aspect. They can be inlined within the aspect using *Inline Interface within Aspect* (16), as well as the connections with their implementing classes, using *Encapsulate Implements with Declare Parents* (5). Sometimes the interface is not an interface at all, but an abstract class, which includes some concrete elements. These definitions can be separated into an aspect using *Split Abstract Class between Aspect and Interface* (21), after which we can use *Change Abstract Class to Interface* (3) to turn the resulting pure abstract class into an interface.

2.1.1 Change Abstract Class to Interface

Typical situation

You have an abstract class that prevents their subclasses from inheriting from some other class.

Recommended action

Turn the abstract class into an interface and change its relationship with the subclasses from inheritance to implementation.

Motivation

This refactoring will be typically used when someone is using *Split Abstract Class between Aspect and Interface* (21) in an abstract class. It comprises the final step of turning it into an interface, to be performed after all concrete members were moved to an aspect. By then the class should be a pure abstract class (i.e. an abstract class without any concrete methods and non-static fields). Of course, we could argue that a pure abstract class should always be turned in an interface, for this makes it plain that we are not in the presence of any concrete elements. This way all implementing classes will be free to extend some other class if there is a need to.

If the abstract class inherits from another class, this refactoring may change type relationships. Depending on the specific relationships, this might break the source code or not, and you must check for potential problems. If you see no problems, start applying this refactoring to the highest class of the inheritance chain and then proceed downwards.

This refactoring will be possible only if all classes in the inheritance chain are owned by the programmer. In addition, if there are concrete classes higher up this refactoring cannot be applied. At the very least, some restructurings in the inheritance chain must be performed first.

Keep in mind that all signatures declared in the resulting interface must be public, and along with it all the corresponding methods of implementing classes.

Mechanics

- Change the keywords 'abstract class' to 'interface'.
- If the abstract class implements some interfaces, change the 'implements' keyword to 'extends'.
- You can optionally remove the keyword 'abstract' from the method declarations.
- Update all classes inheriting from the abstract class. For each subclass, change the keyword from 'extends' to 'implements'. If the subclass implements other interfaces, just remove the 'extends' keyword and move the name of the former abstract class to the list of implemented interfaces.

Example

See also the example for *Split Abstract Class between Aspect and Interface* (21).

```
public abstract class Builder {
   public abstract void processType(String type);
   public abstract void processAttribute(String type);
   public abstract void processValue(String type);
   public abstract String getResult();
}
```

```
public class StructureBuilder extends Builder {
   //...
}
```

```
public class TextBuilder extends Builder {
   //...
}
```

↓

```
public interface Builder {
   public void processType(String type);
   public void processAttribute(String type);
   public void processValue(String type);
   public String getResult();
}
```

```
public class StructureBuilder implements Builder {
   //...
}
```

```
public class TextBuilder implements Builder {
   //...
}
```

2.1.2 Encapsulate Implements with Declare Parents

Typical situation

One or several classes implement an interface related to a concern being modularised into an aspect. The interface represents a role that the classes play only in the context of the aspect.

Recommended action

Replace the 'implements' declaration with a 'declare parents' within the aspect.

Motivation

Interfaces are the standard way in Java to represent the various roles played by a class. AspectJ makes it possible to encapsulate many of these roles within aspects, which often resort to marker interfaces to represent them. It is likely that in the process of moving the role to the aspect all references to the interface will be moved as well. When that happens the interface should also be inlined, using *Inline Interface within Aspect* (16), and turned into a marker interface. The references to the interface include the 'implements' declarations in the various implementing classes.

Mechanics

- Create the suitable 'declare parents' in the aspect.
- In the implementing class, delete the 'implements' clause related to the interface.
- Compile and test.

Example: Moving the First Implements Clause

```
class SomeImplementingClass implements TargetInterface, ... {
   //...
```

↓

```
class SomeImplementingClass implements ... {
   //...
```

```
public aspect Implementation {
   declare parents: SomeImplementingClass implements TargetInterface;
   //...
}
```

Example: Moving the First Implements Clause

When the aspect already has at least one 'declare parents' related to that interface, switch to the following notation:

```
class AnotherClass implements TargetInterface, ... {
   //...
```

↓

```
class AnotherClass implements ... {
   //...
```

```
public aspect Implementation {
   declare parents: (SomeImplementingClass || AnotherClass) implements TargetInterface;
   //...
}
```

2.1.3 Extract Feature into Aspect

Typical situation

The base code includes a feature whose code is scattered across several units of modularity such as methods and classes. You would like to be able to evolve it separately from the primary code base.

Recommended action

Make the feature pluggable by extracting all the related code into an aspect.

Motivation

The most usual elements of implementation of classes are methods and fields, which are the implementation elements we usually move from one module to another when performing refactorings. However, some programming languages provide support for other kinds of elements, such as, in the case of Java, inner classes and inner interfaces. We don't find in the catalogue presented in [8] refactorings for moving these constructs from one class to another. We think the reason for this omission is that it wouldn't make sense to provide refactorings to move them elsewhere. This is due to the nature of these constructs. They comprise implementation elements of classes that are tightly coupled to the rest of the implementation. They are not meant to be part of the interface of modules, and are used instead to provide a better internal structure for the classes. In addition, those elements often make sense only within their current enclosing classes, due to the limitations in Java's composition capabilities. Indeed, these very limitations often motivate the use of inner classes. They help to better separate some parts of a class from other parts, when ideally we would place some of those parts in separate modules[1]. However, when we have AOP's superior composition capabilities we have the option of doing exactly that. With AspectJ, we can design structures that wouldn't be possible with plain Java, and suddenly it makes sense to extract, move and inline certain kinds of members that we wouldn't previously consider, including ones currently represented using inner classes and interfaces.

Some of the existing uses of those constructs will not comprise a good implementation option within aspects, any more than they do within classes. The solution in those cases is to move them to the new modules (i.e. aspects) and then restructure the internals of these modules, using refactorings like *Tidy Up Internal Aspect Structure* (35). Though the issue of restructuring the internals of aspects falls outside the scope of this refactoring, we consider it important to include here a brief mention to the tasks that lie ahead after extracting a concern. The aim is to stress that not everything is done as soon as we move a concern to an aspect.

Mechanics

- Create an empty aspect in the appropriate package.
- If you find inner classes related to the extracted concern use *Extract Inner Class to Standalone* (13) to each of them. You can later use *Inline Class within Aspect* (15).
- Move the concern's various fields to the aspect with *Move Field from Class to Inter-type* (17). Since fields are usually private, you may have to temporarily declare the aspect as privileged in order to keep the code compilable and testable.
- Move initialisation code placed within the constructors using *Extract Fragment into Advice* (9). If some of that code uses some of the constructor's parameters, you should first restructure that part of the code. Consider using *Extract Method* ([8], p.110) to replace the parameter in the constructor and related code with a separate method.

  You can use *Partition Constructor Signature* (43) to deal with cases in which the constructor is part of a published interface that cannot be changed.
- Move the concern's various methods to the aspect with *Move Method from Class to Inter-type* (19).
- When only part of the method relates to the concern we have two options: (1) use *Extract Method* ([8], p.110) and then *Move Method from Class to Inter-type* (19), or (2) use *Extract Fragment into Advice* (9). When the fragment uses an argument from the enclosing method, it may be simpler to use the latter.
- Apply *Inline Class within Aspect* (15) to any former inner classes that are used only within the aspect. Likewise, apply *Inline Interface within Aspect* (16) to any interfaces that are no longer used outside the aspect, and are not expected to in the future.

[1] Eckel's flower example for the Observer pattern [6] is a good example of this: its extensive use of inner classes is meant to compensate for the lack of mixin inheritance [4].

- Change to private the access modes of all aspect members that are now used only within the aspect.
- Remove the qualifier 'privileged' from the aspect as soon as it no longer accesses non-public members in the primary code.

After extracting all the elements from the primary code, consider using *Tidy up Internal Aspect Structure* (35) to improve the internal structure of the resulting aspect.

Example: Extracting Two Concerns from a Tangled Stack

Here we provide a small complete example (initially presented in [14]) that illustrates some of the issues of extracting features from an existing class. It helps to demonstrate how each of these refactorings fit in the larger picture. We do not present a client program, but care was taken to ensure that the refactorings are transparent to any client code.

This example comprises a FIFO structure plus two crosscutting concerns: (1) support to a simple window view of stack's state and (2) precondition checking. This is a case where the responsibility for checking preconditions lies in the client, which explains why the exception used is unchecked[2].

```
import javax.swing.*;

public class TangledStack {
   private int _top = -1;
   private Object[] _elements;
   private final int S_SIZE = 10;
   private JLabel _label = new JLabel("Stack ");
   private JTextField _text = new JTextField(20);

   public TangledStack(JFrame frame) {
      _elements = new Object[S_SIZE];
      frame.getContentPane().add(_label);
      _text.setText("[]");
      frame.getContentPane().add(_text);
   }
   public String toString() {
      StringBuffer result = new StringBuffer("[");
      for(int i=0;i<=_top;i++) {
         result.append(_elements[i].toString());
         if(i!=_top)
            result.append(", ");
      }
      result.append("]");
      return result.toString();
   }
   private void display() {
      _text.setText(toString());
   }
   public void push(Object element) {
      if(isFull())
         throw new PreConditionException("push when stack full.");
      _elements[++_top] = element;
      display();
   }
   public void pop() {
      if(isEmpty())
         throw new PreConditionException("pop when stack empty.");
      _top--;
      display();
   }
   public Object top() {
      if(isEmpty())
         throw new PreConditionException("top when stck empty.");
      return _elements[_top];
   }
   public boolean isFull() {
      return (_top == S_SIZE-1);
   }
   public boolean isEmpty() {
      return (_top<0);
   }
}
```

[2] We do not to present the definition of the runtime exception, as it is quite trivial.

We start by extracting the window view concern from the base code, by applying *Extract Feature into Aspect*. We first create an empty aspect *WindowView* and then move all members related to this concern. These include two fields, *_label* and *_text*, so we start with these, by applying *Move Field from Class to Inter-type* (17) to each in turn.

Both field transfers require similar sequences of steps: (1) copy the declaration of the field to the aspect, (2) add 'TangledStack.' before the field's name, (3) delete (or comment out) the field's original declaration, (4) when moving the first field include the declaration 'import javax.swing.*;' in the import section of the aspect, and (5) change the field's access to public. Compile and test after moving each field.

The initialisation code for both fields should be transferred next. The constructor receives an argument (the JFrame object) related to the extracted concern, so *Partition Constructor Signature* (43) is used. The result is two versions of the constructor, the first – argument less – is placed in the host class and deals only with the remaining concerns (only the primary concern in this case). The other constructor – receiving the JFrame object that is related to the extracted concern – is placed in the aspect, and therefore made pluggable. As this constructor should not include any code unrelated to its concern, it includes a call to super() rather than duplicate the other initialisation code.

We use *Move Method from Class to Inter-type* (19) to move the method display(), and we use *Extract Fragment into Advice* (9) to move the calls to display() in the push() and pop() methods to a piece of advice. The declaration 'import javax.swing.*;' can be removed from the host class at this point. Finally, we can change to private the access qualifiers of the two moved fields and method.

Extraction of the precondition checking concern is similarly performed according to *Extract Feature into Aspect*, though this case is simpler, comprising three executions of *Extract Fragment into Advice* (9) for the tests in push(), pop() and top(), respectively. After both aspects are created as described the host class and the two aspects should look like the following:

```
public class TangledStack {
   private int _top = -1;
   private Object[] _elements;
   private final int S_SIZE = 3;
   public TangledStack() {
      _elements = new Object[S_SIZE];
   }
   public String toString() {
      StringBuffer result = new StringBuffer("[");
      for(int i=0;i<=_top;i++) {
         result.append(_elements[i].toString());
         if(i!=_top)
            result.append(", ");
      }
      result.append("]");
      return result.toString();
   }
   public void push(Object element) {
      _elements[++_top] = element;
   }
   public void pop() {
      _top--;
   }
   public Object top() {
      return _elements[_top];
   }
   public boolean isFull() {
      return (_top == S_SIZE-1);
   }
   public boolean isEmpty() {
      return (_top<0);
   }
}
```

```
import javax.swing.*;

public aspect WindowView {
   private JLabel TangledStack._label =
      new JLabel("Stack ");
   private JTextField TangledStack._text = new JTextField(20);
   public TangledStack.new(JFrame frame) {
      this();
      frame.getContentPane().add(_label);
```

```
      _text.setText("[]");
      frame.getContentPane().add(_text);
   }
   private void TangledStack.display() {
      _text.setText(toString());
   }
   pointcut stateChange(TangledStack stack):
      (execution(public void stack.TangledStack.push(Object))
       ||
       execution(public void stack.TangledStack.pop()))
      && this(stack);
   after(TangledStack _this) returning :
      stateChange(_this) {
      _this.display();
   }
}
```

```
public aspect PreConditionChecking {
   pointcut checkPush(TangledStack stack):
      execution(public void
         TangledStack.push(Object))
      && this(stack);
   before(TangledStack _this): checkPush(_this) {
      if(_this.isFull())
         throw new PreConditionException("push when stack full");
   }
   pointcut checkPop(TangledStack stack):
      execution(public void TangledStack.pop())
      && this(stack);
   before(TangledStack _this): checkPop(_this) {
      if(_this.isEmpty())
         throw new PreConditionException("pop when stack empty");
   }
   pointcut checkTop(TangledStack stack):
      execution(public Object TangledStack.top())
      && this(stack);
   before(TangledStack _this): checkTop(_this) {
      if(_this.isEmpty())
         throw new PreConditionException("top when stack empty");
   }
}
```

2.1.4 Extract Fragment into Advice

Typical situation

Part of a method is related to a concern whose code is being moved to an aspect.

Recommended action

Create a pointcut capturing the required joinpoint and context and move the code fragment to an appropriate advice based on the pointcut.

Motivation

This refactoring should be used when we want to move a piece of functionality to an aspect but we also want it to run in all the places where it presently stands. The functionality to extract does not generally comprise a complete method, sometimes it is a simple method call. Sometimes it is convenient to turn the part that should be moved into its own method, using *Extract Method* ([8], p.110]), and next use *Move Method from Class to Inter-type* (19) on it. However, even in such occasions there will be an advice calling the new method.

Before copying the code fragment a careful analysis of the method's (or constructor's) body should be performed, in order to find a suitable pointcut to capture the exact set of intended joinpoints. If the primary code does not offer a suitable joinpoint, one or more refactorings may have to be performed until the code is ripe for this refactoring.

Sometimes the advice will need to capture local variables (either primitives or object references). These are a problem, because AspectJ cannot capture the values of local variables. These situations may be a sign that the method is more complicated than it should be. Consider whether it would make sense to split it in various parts, using *Extract Method* ([8], p.110]) for each part in turn. Such a split may provide the joinpoints you need. For instance, the arguments or return value of one of extracted methods may expose the context that was previously available only in a local variable.

In the more awkward cases when even the above options are of no avail, use *Replace Method with Method Object* ([8], p.135). This is the refactoring recommended by Fowler et al to ease the way for *Extract Method* ([8], p.110), but it may be even more appropriate to the present case, for it is almost certain to provide you with the missing leverage for context capture. This solution is preferable to crudely turning the local variable into a field: the lifetime of fields of the method object is restricted to the execution of the method, rather than to the whole lifetime of the host object. However, keep in mind that it may not be possible to keep the fields of the method object private: consider using *Encapsulate Field* ([8], p.206). In addition, *Replace Method with Method Object* ([8], p.135) cannot be used in constructors and recursive methods.

If the fragment to be extracted uses an internal type, consider first using *Extract Inner Class to Standalone* (13) on that type before applying this refactoring.

Mechanics

- Create a named pointcut that captures the intended set of joinpoints. If the intended pointcut already exists (from previous uses of *Extract Fragment into Advice*), change it so that it includes the joinpoint related to the present fragment.

- Ensure that the pointcut also captures all context required by the code fragment. In particular, check if the extracted fragment mentions 'this' or 'super', or includes self-calls. In such cases, a reference to the executing object should be captured. Choose a suitable name for the variable holding the captured object. In some cases, the choice may be straightforward. In others, use a general yet meaningful name such as '_this' or 'self'.
  Example:

```
pointcut stateChange(TangledStack stack):
     execution(public void TangledStack.push(Object))
     && this(stack);
after(TangledStack _this) returning : stateChange(_this) {
     _this.display();
}
```

- Check whether all types used in the pointcut are known to the aspect. In some cases new import declarations may need to be added (this applies even when the type is mentioned only in pointcuts).

- Create the suitable advice for the pointcut, with an empty body (in case it is not already under construction).

- Move the code to extract from the source method into the advice's body.

- Replace references to the self-variable 'this' by the corresponding variable obtained from the context capture.

- Scan the extracted code for references to any variables that are local in scope to the source method, including parameters and local variables. Declarations of any temporary variables used only within the extracted code can be placed inside the advice's body.
  You may need to make some adjustments to set up the advice's context in some cases.

When the advice is meant to replace a large number of scattered fragments you should choose the simpler of the two options: (1) to deal with the whole set at a single go, or (2) to deal with one fragment at a time. Sometimes the pointcut is complicated to specify when covering only a subset of the intended joinpoints. If that is the case, you may consider writing the full, intended pointcut right at the start. The drawback then is that you'll have to factor all the scattered fragments to the common advice at a single go before you can compile and test again. You should avoid this whenever the scattered fragments are not identical or very similar (e.g. calls to the same method). In some cases, it may be worthwhile to refactor the various fragments so that they become more alike (e.g. giving the same names to locals and parameters) and therefore easier to reason with.

Example: Simple Extraction of a Method Call

```
public class TangledStack {
   //...
   public void push(Object element) {
      _elements[++_top] = element;
      display();
   }
```

↓

```
pointcut stateChange(TangledStack stack):
   execution(public void TangledStack.push(Object))
   && this(stack);

after(TangledStack _this) returning :
   stateChange(_this) {
   _this.display();
}
```

Example: Complex Example Requiring Refactoring to Make the Code Aspect-friendly

The following case arose during the refactoring experiment described in [14], in which we extracted all code related to a concern into an aspect, in order to make it pluggable. At a given time, we wanted to use *Extract Fragment into Advice* on the RepeatUntilProcedure.compile() method but the extracted code needed various local values from the previous part of the method for its computation. The values were stored neither as fields of some object nor as the return value. In addition, the method was overlong. The structure of the code was as sketched below:

```
public class RepeatUntilProcedure ... {
   //...
   public Graph compile() {
      // 17 LoCs
      if (...)
         // one LoC
      else {
         //13 LoCs related to the primary concern
         //66 LoCs related to the data link concern
      }
      //a few more LoCs related to the data link concern
   }
   return result;
}
```

```
public aspect DataLinksAspect {
   //...
}
```

Using *Extract Method* ([8], 110) on the chunk of code of interest, we make it easier to reason with.

```
public class RepeatUntilProcedure ... {
   //...
   private boolean compileDataLinks(<various arguments>) {
      // 66 LoCs
      return result;
   }
   public Graph compile() {
      // 17 LoCs
      if (...) {
         //1 LoC to prepare for the call to compileDataLinks()
         ... = compileDataLinks(...);
         //A few more LoCs
      }
      return result;
   }
```

Next, we used *Move Method from Class to Inter-type* (19) to the compileDataLinks() method:

```
public class RepeatUntilProcedure ... {
   //...
   public Graph compile() {
      // 17 LoCs
      if (...) {
         //1 LoC to prepare for the call to compileDataLinks()
         ... = compileDataLinks(...);
         //A few more LoCs
      }
      return result;
   }
```

```
public aspect DataLinksAspect {
   //...
   //private
   public
   boolean RepeatUntilProcedure.compileDataLinks(...) {
      // 66 LoCs
      return result;
   }
}
```

We then applied *Replace Method with Method Object* ([8], p.135) to compileDataLinks(). The class of the method object was implemented with the Compile inner class within RepeatUntilProcedure:

```
public class RepeatUntilProcedure ... {
   //...
   public class Compile {
      private final RepeatUntilProcedure _enclosing;
      //several fields stemming from the local variables of compileDataLinks()
      public Compile(RepeatUntilProcedure repeatUntilProc) {
         _enclosing = repeatUntilProc;
      }
      public Graph compute() {
         //The same 17 LoCs of compile(), with the following differences:
         // ° the code uses the fields instead of the former locals
         // ° some references to this were replaced by _enclosing
         if (...) {
            //1 LoC to prepare for the call to compileDataLinks()
            ... = compileDataLinks(...);
            //A few more LoCs related to the data link concern
         }
         return _result;
      }
   }
   //...
```

We were finally ready to apply *Extract Fragment into Advice*, leading to the following layout:

```
public class RepeatUntilProcedure ... {
   //...
   public class Compile {
      //...
      public Graph compute() {
         //17 LoCs
         return _result;
      }
   }
   //...
```

```
public aspect DataLinksAspect {
   //...
   private boolean compileDataLinks(<various arguments>) {
      // 66 LoCs
      return result;
   }
   pointcut repeatUntilProcComputeCompile(
         RepeatUntilProcedure _this,
         RepeatUntilProcedure.Compile compile):
      call(public Graph RepeatUntilProcedure.Compile.compute())
      && target(compile)
      && this(_this);

   /** Add processing of data-links at the end of the RepeatUntilProcedure compile. */
   after(RepeatUntilProcedure _this, RepeatUntilProcedure.Compile compile)
         returning: repeatUntilProcComputeCompile(_this, compile) {
      if (...) {
         //1 LoC to prepare for the call to compileDataLinks()
         ... = _this.compileDataLinks(compile, hasFeedBack);
         //A few more LoCs related to the data link concern
      }
   }
}
```

The refactoring is complete. Notice that compileDataLinks() could be made private again, this time to the aspect. In circumstances such as these, the code inside the aspect is likely to need some tidying up.

2.1.5 Extract Inner Class to Standalone

Typical situation

An inner class relates to a concern that is being extracted into an aspect.

Recommended action

Eliminate all dependencies from private members of the enclosing class and provide it with a reference to the enclosing object. The inner class can then become standalone.

Motivation

Even if Java programmers are unable to decouple a class from certain accessory functionalities, they can at least place those functionalities in a well-localised way, separate from the rest of the class's code. Inner classes can be used to structure the internals of a class and localise functionalities not related to the primary functionality of the class, thereby separating it from the rest of the class's implementation. This is possible because inner classes can refer to any of the members of the enclosing object, including the private ones. To a limited extent, inner classes can be used as if they were subclasses of its enclosing class, with the advantage that they can still inherit from another unrelated class.

However, when we have AOP's superior composition capabilities this may no longer be the best available option. The kinds of functionality placed within inner classes are best modularised within aspects, and that in turn provides the motivation for extracting inner classes from their enclosing classes, something that would probably not make sense with plain Java.

Turning inner classes into standalone classes is a preparatory step before moving them to within the aspect, using *Inline Class within Aspect* (15). In addition, the limitation of Java's composition capabilities makes it likely that the functionality placed in the extracted classes is duplicated in multiple classes. In such cases, applying this refactoring will expose that duplication, particularly when several inner classes have the same name. Placing the various elements of the crosscutting concern within an aspect is very effective to expose various kinds of duplications, including of this kind.

Mechanics

- Look for any code within the inner class relating to behaviour that should be kept within the enclosing class. Use *Extract Method* ([8], p.110) on those parts.
- Create in the inner class a private field of the type of the enclosing class.

  In some cases, the definition of the inner class is repeated within several enclosing classes, which may not be related by a common type relationship. In such cases first create an interface exposing the common interface used within the inner class, make the enclosing classes implement that interface. Make the inner class's private field of that interface type. This way, a common inner class can be extracted from all the enclosing classes. If you intend to inline the inner classes within an aspect, the interface should be inlined as well – using *Inline Interface within Aspect* (16) – after all its implementing classes.
- Create a public constructor for the inner class and include a parameter for the enclosing object providing the initial value of the new field. Update any code related with the creation of instances of the inner class in the enclosing class. If the inner class is not private, also check possible uses outside the enclosing class.
- Compile and test.
- Look for all direct references to fields belonging to the enclosing object. Use *Self Encapsulate Field* ([8], p.171) on all such fields.
- Look for any calls to private methods made within the inner class. Relax the access rules of those methods.

If you are reluctant to expose those members, you still have the option of doing it only on a temporary basis. As soon as the class is inlined within an aspect, reclassify them as private. However, in this case the aspect will have to be privileged.

You can make sure that no members are forgotten by placing 'this**.**' before each field reference and method call (Java does not allow the use of 'this' within inner classes to refer to members of the enclosing class). This is easy to do for inner classes, since they are typically not very large (otherwise that could be a case for applying *Extract Class* [8], p.149).

- Compile and test.
- Create a standalone class with the same name as the inner class. Copy the source text of the inner class to the standalone class and add 'public' before the class's name. Check for imports that should go along with the class, as well as imports that the host class no longer needs. Delete the inner class.
- Compile and test.

Example

The following example uses a simplified version of the Flower class from Eckel's flower example for the Observer design pattern [6]. The purpose is to turn the inner class OpenNotifier into a standalone class.

```
public class Flower {
   private boolean _isOpen;
   private OpenNotifier _oNotify = new OpenNotifier();

   public Flower() {
      _isOpen = false;
   }
   public void open() { // Opens its petals
      System.out.println("Flower open.");
      _isOpen = true;
      _oNotify.notifyObservers();
   }
   public Observable opening() {
      return _oNotify;
   }
   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;
      }
   }
```

First, create the field _enclosing and a constructor receiving an argument to initialise that field. The creation of an OpenNotifier object needs to be updated (the compiler can be very useful in this kind of situations):

```
public class OpenNotifier extends Observable {
   private Flower _enclosing;
   private boolean _alreadyOpen = false;
   public OpenNotifier(Flower enclosing) {
      this._enclosing = enclosing;
   }
   public void notifyObservers() { //...
```

```
public class Flower {
   private boolean _isOpen;
   private OpenNotifier _oNotify = new OpenNotifier(this);
```

Next, replace direct references to fields from the enclosing class. There is one in this example, _isOpen:

```
   boolean isOpen() {
      return this._isOpen;
   }

private class OpenNotifier extends Observable {
   private Flower _enclosing;
   private boolean _alreadyOpen = false;
   public OpenNotifier(Flower enclosing) {
      _enclosing = enclosing;
   }
   public void notifyObservers() {
      if(_enclosing.isOpen() && !_alreadyOpen) {
```

Next, ensure that all references to member within the inner class have either 'this.', 'super.' or '_enclosing.' (you could also use 'this._enclosing.', of course):

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

Finally, create a standalone class with the same name, copy the contents and delete the inner class.

```
import java.util.Observable;

public class OpenNotifier extends Observable {
   //...
```

2.1.6 Inline Class within Aspect

Typical situation

A small standalone class is used only by code within an aspect.

Recommended action

Move the class to within the body of the aspect.

Motivation

This situation can occur when one or several small helper classes relate to a concern that is being modularised into an aspect. It also occurs during the process of moving an inner class from within a class to within an aspect, after applying *Extract Inner Class to Standalone* (13).

Inner classes are typically small and this refactoring is recommended only for small classes (as a rule of thumb, a class with more than a screenful of lines of code may be a sign that it should remain standalone).

Mechanics

- Create a copy of the class's source code, inside the aspect. Replace public with static just before the class's name.
- Add in the aspect's import section any imports that may be needed by the class.
- Remove the standalone class.
- Compile and test.

Example

We take the example of the class OpenNotifier, from Eckel's flower example for the Observer design pattern [6]. We start at the point after OpenNotifier was made a standalone class.

```
import java.util.Observable;

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

The class can be copied almost "as is" to within the body of the aspect, only the first keyword being changed:

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

In this case, code using OpenNotifier needs to import java.util.Observable. If the aspect does not include that import already, it must be added as well. After compiling and testing, this refactoring is complete.

2.1.7 Inline Interface within Aspect

Typical situation

One or several interfaces are used only within an aspect.

Recommended action

Turn the interfaces into inner interfaces placed within the aspect.

Motivation

This situation usually arises when a feature is being extracted from the existing code base. Often, the feature assigns various roles to various participants using an interface to model each role. As related code is being moved to an aspect, at some point only the aspect knows about the interfaces. In such cases, there is no compelling reason for keeping them as standalone.

The most desirable situation is to inline one interface at a time, leaving the code in a compilable state and passing all the tests before inlining the next interface. However, sometimes the system relies on a set of interdependent interfaces with each interface declaring methods that refer to other interfaces in their parameter lists. In such cases it may be easier to move the whole set of interfaces at one go than to move one at a time.

Mechanics

- Create within the aspect a private copy of the standalone interface.
- Delete the standalone interface[3].
- Compile and test.

Example

In the following example, a Mediator[4] aspect is being created to encapsulate the interaction between various Button objects. Interfaces GUIColleague and GUIMediator are used only within the Mediator aspect. Moving only one of the interfaces leads to compiler errors, so they are moved at one go.

```
public interface GUIMediator {
   public void colleagueChanged(GUIColleague colleague);
}
```

```
public interface GUIColleague {
   public void setMediator(GUIMediator mediator);
}
```

```
public aspect Mediator {
   declare parents: Button implements GUIColleague;
   declare parents: Label implements GUIMediator;
   //...
```

➜

```
public aspect Mediator {
   private interface GUIMediator {
      public void colleagueChanged(GUIColleague colleague);
   }
   private interface GUIColleague {
      public void setMediator(GUIMediator mediator);
   }
   //...
```

2.1.8 Move Field from Class to Inter-type

Typical situation

A field relates to a concern other than the primary concern. An aspect encapsulating the secondary concern is under construction, which will harbour all the concern's code.

Recommended action

Move the field from the class to the aspect as an inter-type declaration.

Motivation

If the field is of an internal type, use *Extract Inner Class to Standalone* (13) on that type before applying this refactoring.

Mechanics

- Copy the declaration of the field from the class to the aspect, including the assignment of an initial value, if one exists. Add the host class's name and '**.**' before the name of the field in the inter-type declaration.
- Check whether a new *import* statement should be written in the aspect's *import* section, to bring the field's type into its scope. If the aspect is placed in a separate package, check also for the declaration of the host class.

---

[3] To play safe, make sure that *.class files are not left in the binary directory.

[4] The code used in this example was taken from the Mediator example of [11].

- If the field's access is *private,* change it to a less restrictive one. If the aspect is placed within the same package as the host class it can be *package-protected*, otherwise it must be *public*. You will be able change it back to *private* as soon as all code dealing with the field is placed in the aspect.
- Delete the field's declaration in the host class. Check for any *import* statements that are no longer necessary in the original host class.
- Check for pointcuts using the within() pointcut designator referring to the moved field.
- Compile and test.
- Create a 'declare warning' to signal all occurrences of a given member in the code, as shown in the following example:

```
public aspect ThisAspect {
     //...
     declare warning:
        (get(JTextField TargetClass._text) || set(JTextField TargetClass._text))
        && !within(ThisAspect):
        "Field _text is accessed outside aspect.";
     //...
}
```

- For each fragment of code using the field, decide whether the whole method, or just a fragment, should be moved to the aspect: (a) if the whole method must be moved, use *Move Method from Class to Inter-type* (19). (b) If just a fragment should be moved, use *Extract Fragment into Advice* (9). (c) If a parameter of a method or constructor relates to the moved field use *Extract Method* ([8], p.110) to isolate the part that uses the parameter, move it to the aspect with *Move Method from Class to Inter-type* (19), move the call to a more suitable point, and then use *Remove Parameter* ([8], p.277). Use *Replace Inter-type Method with Aspect Method* (33) to deal with inter-type methods that use the inter-type fields. Compile and test after each refactoring.
- As soon as the last access to the field outside the intended scope is removed, restrict again the field's access. This is usually means *private*, but if for some reason you need to keep in the class some code related to the field, consider using *Encapsulate Field* ([8], p.206). If the field was originally *protected* and subclasses of the target class need to access it, use *Introduce Aspect Protection* (26).

Example

```
//...
import javax.swing.*;

public class TangledStack {
   private int _top = -1;
   private Object[] elements;
   private final int S_SIZE = 10;
   private JLabel _label = new JLabel("Stack ");
   private JTextField _text = new JTextField(20);
   //...
}
```

↓

```
public class TangledStack {
   private int _top = -1;
   private Object[] elements;
   private final int S_SIZE = 10;
   //...
}
```

```
import javax.swing.*;

public aspect WindowView {
   public JLabel TangledStack._label = new JLabel("Stack");
   public JTextField TangledStack._text = new JTextField(20);
   //...
}
```

When all the code relative to the fields is placed in the aspect, change their access qualifiers back to private, compile, and test again:

```
import javax.swing.*;

public aspect WindowView {
   private JLabel TangledStack._label = new JLabel("Stack");
   private JTextField TangledStack._text = new JTextField(20);
   //...
}
```

2.1.9 Move Method from Class to Inter-type

Typical situation

A method in a class belongs to a concern other than the primary concern.

Recommended action

Move the method into the aspect that addresses the secondary concern, as an inter-type declaration.

Motivation

If the method contains some logic that should remain in the class, first apply *Extract Method* ([8], p.110).

If the method uses an internal type for its return value or any of its arguments, use *Extract Inner Class to Standalone* (13) on that type before applying this refactoring.

The most straightforward case of moving a method to an aspect is when the method is public, there is only one implementation of its signature throughout the inheritance chain, and it uses only (1) its parameters, (2) public members, (3) local variables, (4) members already moved from the class to the aspect which are (perhaps temporarily) qualified as public. If these conditions are not all met, some of the following cases should be considered.

a) Check if the method uses any non-public member that may not be visible in the aspect. Consider whether these should also belong to the aspect's concern. If you think they belong to the aspect, consider whether they would be best moved together or one at a time. In some cases, several members may be tightly coupled and would be easier to move together. In case you want to move them one at a time, start with the fields, applying *Move Field from Class to Inter-type* (17), next move initialisation code in the constructors[5] with *Extract Fragment into Advice* (9), and then move the methods with this refactoring.

b) If the method uses non-public members that you think should remain in the class, check if there are public accessor methods you can use, or if it is worth to create them now, even if just temporarily, or if you can change their accesses to a less restrictive one. See also if it is a case of moving the aspect to the same package. If you are reluctant to use any of these options, you'll have to declare the aspect as privileged.

c) A situation where a method needs to access non-public members in both the host class and the aspect may be an indication that the method is addressing more than one concern. If it is the case, the best solution is probably to leave the method in the class, keeping the code relative to the main functionality, and moving the part related to the target concern, using *Extract Fragment into Advice* (9).

d) If the moved method is non-public see if you can move all the methods that call the moved method also belong to the same concern, the same way as in a). Of course, this is feasible only when just a few methods and fields are involved.

[5] The refactoring *Partition Constructor Signature* was designed for a special case that can arise here, if the constructor is part of a published interface that the developers are unable or unwilling to modify.

e) Search for any implementations of the same signature in sub- and super-classes. In case you find some, these alternative implementations should belong to the aspect as well, in order to make the related functionality pluggable.

f) A full inheritance hierarchy in the primary code may be a sign that the concern already aligns well with the dominant decomposition. Check if that is the case, or whether it would not be better to leave the hierarchy in the primary code and extract only a subset of the code of each of the implementations, using *Extract Fragment into Advice* (9).

Apply *Move Method from Class to Inter-type* to each of the alternative implementations in turn. Start with the implementations in the leaf classes, and then move up the inheritance hierarchy.

In the case the access of the methods is protected you may need to change them to public, especially if the aspect is placed on a different package than the class. As soon as all the implementations are in the aspect, see if you can change the accesses to private. If you can't, leave the access as public and use *Introduce Aspect Protection* (26).

Mechanics

- Copy the method's definition to the aspect. Add the class name and '.' before the name of the method.
- If the access is non-public, it may need to be (temporarily) adjusted to a less restrictive access. If the aspect is placed within the same package as the host class, package-protected is enough. Otherwise, only public will do.
- Check whether a new import statement should be written in the aspect's import section. Check also whether any import statement is no longer necessary in the host class.
- Check for pointcuts referring to the moved method using the within() pointcut designator.
- Delete the method's definition in the class.
- Compile and test.
- The following steps apply if you want to make the method private to the aspect. Add a 'declare warning' to signal all calls to the method, as shown in the following example:

```
public aspect ThisAspect {
     //...
     declare warning:
         (call(<type> <host class>.someMethod(<arguments>))
         && !within(ThisAspect):
         "method <host class>.someMethod() is called outside ThisAspect.";
     //...
}
```

- As soon as all the code that uses the method is in the aspect, change it to private. In case of protected access leave a 'declare error' as prescribed in *Introduce Aspect Protection* (26).

Example: moving a private method.

```
public class TangledStack {
   private void display() {
       _text.setText(toString());
   }
   //...
}
```

↓

```
public class TangledStack {
   //...
}
```

```
public aspect WindowView {
   //...
   public //private
   void TangledStack.display() {
       _text.setText(toString());
   }
}
```

Apply *Move Method from Class to Inter-type* first to the "leaf-methods" (i.e. the methods of classes placed lower in the inheritance hierarchy), as they will betray fewer dependencies. Then progressively apply the refactoring up the inheritance chain until you reach the top method.

2.1.10 Split Abstract Class between Aspect and Interface

Typical situation

You have one or several classes inheriting from an abstract class, which prevents them from inheriting from another class. You cannot turn the abstract class into an interface because it defines concrete members.

Recommended action

Move all concrete members defined in the abstract class to an aspect. You can then turn the abstract class into an interface.

Motivation

All classes implementing the interface will inherit the members introduced by the aspect and still be able to inherit from another class.

Mechanics

- Create an aspect that will harbour the concrete members of the abstract class.
- Use *Move Field from Class to Inter-type* (17) to move each field in turn from the abstract class to the aspect.
- When the class includes constructors, you may find two different situations, depending on whether the initialisations use or not values passed to the constructor. If no constructor arguments are required, use *Extract Fragment into Aspect* to move initialisation code from the class's constructor(s) to the aspect. If the initialisation requires arguments, the best option is probably to create setter methods for the fields that need external values and refactor the code so that it uses the setter methods instead of constructors.

  Introducing a constructor from the aspect is not an option because the class is going to be turned into an interface and therefore subclasses will loose the inheritance relationship in the process, as well as the ability to have constructors.
- For each concrete method, use *Move Method from Class to Intertype* to move it from the abstract class to the aspect, keeping in the class an abstract declaration the method's signature.
- Use *Change Abstract Class to Interface* (3) to turn the abstract class into an interface and update the subclasses.

Example

The following example is the Java implementation of the *Factory Method* pattern ([9], p.107) by Hannemann and Kiczales [11]. It is worth it to reproduce some comments found in the code regarding this implementation:

> In this example, the factory method createComponent creates a JComponent (a button and a label, respectively). The anOperation() method showFrame() uses the factory method to show a little GUI. In one case, the created frame contains a button, in the other a simple label.
>
> Since the anOperation() method requires an implementation, Creator has to be an abstract class (as opposed to an interface). Consequently, all ConcreteCreators have to be subclasses of that class and cannot belong to a different inheritance hierarchy.

```
import java.awt.Point;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JComponent;
import javax.swing.JFrame;
```

```
import javax.swing.JPanel;

public abstract class Creator {
   private static Point lastFrameLocation = new Point(0, 0);
   public abstract JComponent createComponent();
   public abstract String getTitle();
   public final void showFrame() {
      JFrame frame = new JFrame(getTitle());
      frame.addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) { System.exit(0); }
      });
      JPanel panel = new JPanel();
      panel.add(createComponent());
      frame.getContentPane().add(panel);
      frame.pack();
      frame.setLocation(lastFrameLocation);
      lastFrameLocation.translate(75, 75);
      frame.setVisible(true);
   }
}
```

First, we create the blank aspect CreatorImplementation. Next, we use *Move Field from Class to Inter-type* (17) to move the field to aspect.

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public abstract class Creator {
   public abstract JComponent createComponent();
   public abstract String getTitle();
   public final void showFrame() {
   //...
   }
}
```

```
import java.awt.Point;

public aspect CreatorImplementation {
   //private
   static Point Creator.lastFrameLocation = new Point(0, 0);
}
```

We then use *Move Field from Class to Inter-type* (17) to move the lastFrameLocation field, which was temporarily turned package-protected. Next, we use *Move Method from Class to Inter-type* (19) to move the showFrame() method, but in this case leaving in the abstract class an abstract declaration of the signature. The lastFrameLocation field can be private again.

```
import javax.swing.JComponent;

public abstract class Creator {
    public abstract JComponent createComponent();
    public abstract String getTitle();
    public abstract void showFrame();
}
```

```
import java.awt.Point;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public aspect CreatorImplementation {
    //private
    static Point Creator.lastFrameLocation = new Point(0, 0);
    public final void Creator.showFrame() {
        //...
    }
}
```

Next, we use *Change Abstract Class to Interface* (3) to turn Creator into an interface. This entails removing the static classification of the lastFrameLocation field. In many cases, this is not behaviour preserving, and it would require some case-specific workaround. As this isn't much of a problem in this particular case, we leave it as shown:

```
import javax.swing.JComponent;

public interface Creator {
   public JComponent createComponent();
   public String getTitle();
   public void showFrame();
}
```

```
import java.awt.Point;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JFrame;
import javax.swing.JPanel;

public aspect CreatorImplementation {
   private //static
   Point Creator.lastFrameLocation = new Point(0, 0);
   public final void Creator.showFrame() {
      //...
   }
}
```

2.2. Restructuring the Internals of Aspects

The main refactoring of this group is *Tidy Up Internal Aspect Structure* (35), dealing with the task of improving the internal structure of an aspect after all elements from a crosscutting concern were moved into it. This refactoring was motivated by one important finding of our research: that aspects resulting from feature extractions are generally badly formed, betraying much duplication and inadequate internal structure. There is duplication because programmers are forced to duplicate code in multiple classes, since they cannot modularise the related concern. When these duplicated elements are moved to aspects (using the refactorings presented in the previous sections) the duplication is not eliminated, just moved to inside the aspect. Even so, moving the elements to an aspect is advantageous because this makes it much easier to deal with duplication. Extracted aspects usually betray an excessive reliance on inter-type declarations (because moving members from the base code to aspects is easier to do using them). When all the members are within the aspect, it is time to check whether they should stay as they are or should they be replaced by a different mechanism.

Our experience showed that applying *Generalise Target Type with Marker Interface* (24) is an effective way to expose and remove various kinds of duplication in inter-type declarations. We also recommend it as a starting step to tidying up the internal structure of aspects resulting from extractions. When using *Generalise Target Type with Marker Interface* (24) it may be expedient to use *Extend Marker Interface with Signature* (24) as a stopgap in some particular cases when the generally applicable code is being separated from case-specific code.

Sometimes we notice that an aspect introduces members to classes when those would ideally be composed more dynamically and flexibly. In such cases we prescribe the use of *Replace Inter-type Field with Aspect Map* (28) and *Replace Inter-type Method with Aspect Method* (33) to replace the introductions with a different logic providing the functionality with the desired advantages. *Introduce Aspect Protection* (26) is used to restore some protection to members that were turned public in the process of being moved to an aspect.

We sometimes find that an aspect contains both generically applicable code and case-specific code. The former can potentially be placed in a reusable superaspect, in which case we use *Extract Superaspect* (36), which in turn prescribes the various *Pull Up* refactorings. Our experience showed that we sometimes need to reverse some of those pulls, and that was our motivation for the *Push Down* refactorings.

2.2.1 Extend Marker Interface with Signature

Typical situation

An inner interface represents a role used only within the aspect. You want to refer to a method specific to the type of an implementing class, but you can't because its signature is not declared by the interface.

Recommended action

Extend the interface with the signature by adding an inter-type abstract declaration of the type-specific signature. You can then proceed with tidying up of the aspect's structure.

Motivation

Sometimes you would like to temporarily resolve that reference because that would enable you to do some tidying up of the aspect's internals, after which you will be well positioned to better deal with that dependence. *Extend Marker Interface with Signature* can be used as a stopgap in such situations to temporarily resolve a dependency. One case in which this situation arises often is during the use of *Generalise Target Type with Marker Interface* (24).

An alternative solution to these problems would be to resort to downcasts. However, downcasts create dependencies to the cast's target type. This type must be included in the aspect's import section and the type's binary file must be available when doing a build. *Extend Marker Interface with Signature* is a superior solution because it avoids such dependencies. The dependency it creates is restricted to a method signature only, not to specific types. For these reasons, this refactoring may be worth using even in (simple) cases when you do not expect the dependency to the type-specific signature to go away after you perform your planned refactorings.

The signature must be public in order to be acceptable to the compiler. In addition, this solution is feasible only if all the types made to implement the marker interface export the signature.

Mechanics

- If the method is not public, change it to public.
- Create in the aspect an inter-type abstract declaration of the method's signature targeting the marker interface that will be used in place of the specific type.
- Compile and test.

Example

The ExampleAspect aspect uses the Role marker interface. Some point in the code using Role resorts to a downcast to specific type SpecificType, to resolve the call to the doSoemthing() method, that is specific to this type. By using *Extend Marker Interface with Signature* (24) we eliminate this dependency to SpecificType. If this is the only use of SpecificType within ExampleAspect, even the import can be removed.

```
import ...SpecificType;

public aspect ExampleAspect {
   private interface Role { }
   ... action(Role obj) {
      //...
      ((SpecificType)obj).doSomething()
```

↓

```
public aspect ExampleAspect {
   private interface Role { }
   public abstract void Role.doSomething();
   //...
   obj.doSomething()
```

2.2.2 Generalise Target Type with Marker Interface

Typical situation

An aspect refers to specific concrete types, preventing it from being reused and reusable.

Recommended action

Replace the references to specific types with a marker interface and make the target types implement the marker interface.

Motivation

This refactoring contributes to reduce the coupling between an aspect and its target code bases. It can also be used to expose and eliminate much duplication that couldn't be eliminated in code referring to specific types. It can also be useful when we want to apply *Extract Superaspect* (36) to aspects containing providing similar functionality, by rationalising their internal structures.

Several situations can prevent *Extract Superaspect* (36) from being immediately applied. One is when the concrete aspects contain both generally applicable code and code specific to concrete classes, and these are tangled, with specific types being used in places where a general marker interface could be used instead. You can use *Generalise Target Type with Marker Interface* to obtain more reusable code, by replacing the specific types with references to a generally applicable marker interface in all points where this can be done. After applying this refactoring, the resulting marker interfaces are the primary candidates for pulling up to a superaspect.

Mechanics

- Create a marker interface representing the role played by the target classes. Create the 'declare parents' to associate the concrete classes to the role.
- Replace the references to the class with references to the marker interface. In cases when the aspect introduces the same field or method to more than one class, replace them with a single introduction to the interface.

  Sometimes the replacement cannot be made in method bodies because parts of the code depend on elements specific to a concrete class. In such cases, consider using *Extract Method* ([8], p.110) to separate the parts covered by the role interface from the parts specific to particular classes. This may be an indication that in future the aspect should be split into a generally applicable abstract superaspect and one or several specific concrete subaspects, using *Extract Superaspect* (36).
- Compile and test.
- When all method introductions refer to the interface, it is possible to remove the declarations of operations (methods) within the interface (if the interface is a marker interface, nested within the aspect, the related operations are defined within the aspect anyway, so removing the declarations from the interface will result in simpler code). If, however, the interface is kept standalone, leave the declarations in place. This way the code will be easier to understand.

Example: Simple Replacements

In the following example, GUIColleague is an interface, representing a role. The aspect Mediator assigns the GUIColleague role to the Button class, but some parts of the code still specifically refer to Button instead of GUIColleague. We want to make all code to depend only on the interface.

```
public aspect Mediator {
   declare parents: Button implements GUIColleague;
   declare parents: Label implements GUIMediator;
   GUIMediator Button._mediator;
   public void Button.setMediator(GUIMediator mediator) {
      this._mediator = mediator;
   }
   pointcut buttonClicked(Button button):
      execution(public void clicked()) && this(button);
   after(Button button): buttonClicked(button) {
```

```
      button._mediator.colleagueChanged(button);
   }
   //...
}
```

↓

```
public aspect Mediator {
   declare parents: Button implements GUIColleague;
   declare parents: Label implements GUIMediator;
   GUIMediator GUIColleague._mediator;
   public void GUIColleague.setMediator(GUIMediator mediator) {
      this._mediator = mediator;
   }
   pointcut buttonClicked(GUIColleague button):
      execution(public void clicked()) && this(button);
   after(GUIColleague button): buttonClicked(button) {
      button._mediator.colleagueChanged(button);
   }
   //...
}
```

Naturally, the names of some variables (such as button) should now be renamed to reflect their more general context.

Example: Eliminating Duplication

In this example aspect, ObservingOpen encapsulates an observing relationship that was just extracted from participant classes. ObservingOpen introduces some fields and methods into several classes playing the Observer role (in this case Bee and Hummingbird). The classes are the only difference between these introductions, so applying *Generalise Target Type with Marker Interface* creates the Subject marker interface and removes the duplication.

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

↓

```
public aspect ObservingOpen ... {
   //...
   private interface Subject { }
   declare parents: (Bee || Hummingbird) implements Subject;
   private OpenObserver Subject.openObsrv = new OpenObserver(this);

   public java.util.Observer Subject.openObserver() {
      return openObsrv;
   }
}
```

2.2.3 Introduce Aspect Protection

Typical situation

A public inter-type member should be visible in an aspect al all its subaspects, but not outside the aspect inheritance chain.

Recommended action

Declare the inter-type member as public and add a ‘declare error’ to prevent its use outside the aspect inheritance chain.

Motivation

AspectJ does not allow the protected access on inter-type members, so whenever we would like to extend its access to subaspects we must classify the member as public. Fortunately the ‘declare error’ mechanism enables us to emulate that protection.

Mechanics

- Add a 'declare warning' in the aspect enclosing the inter-type member, specifying the intended restriction on its use.
- Compile and test.
- For every warning generated by the compiler, perform the refactorings necessary to move the member to the permitted zone of code.
- When there are no more warnings change the 'declare warning' to 'declare error'.

Example: protecting an inter-type field

Consider an abstract superaspect GeneralPolicy declaring inter-type the field _sensitiveData. We want to restrict use of the field to aspect and its subaspects.

```
abstract aspect GeneralPolicy {
   protected interface Participant {}
   public Data Participant._sensitiveData;
   //...
}
```

```
aspect ConcretePolicy extends GeneralPolicy {
   //code using Participant._sensitiveData
}
```

We can add in the superaspect the following 'declare warning':

```
abstract aspect GeneralPolicy {
   protected interface Participant {}
   public Data Participant._sensitiveData;
   declare warning:
      (set(public Data Participant+._sensitiveData) ||
      get(public Data Participant+._sensitiveData))
      && !within(GeneralPolicy+):
      "field _sensitiveData is aspect protected. Not visible here.";
   //...
}
```

After all warnings are gone, we change the 'declare warning' to 'declare error'.

Example: protecting an inter-type method

Suppose the same abstract aspect as in the previous example also includes method processSensitiveData(), which we also would like to protect:

```
abstract aspect GeneralPolicy {
   protected interface Participant {}
   public Data Participant._sensitiveData;
   public void processSensitiveData() {
      //code using Participant._sensitiveData
   }
   //...
}
```

We create the following 'declare warning':

```
abstract aspect GeneralPolicy {
   protected interface Participant {}
   public Data Participant._sensitiveData;
   public void processSensitiveData() {
      //code using caspule._sensitiveData
   }
   declare warning:
      call(void processSensitiveData())
      && !within(GeneralPolicy+):
      "method processSensitiveData is aspect protected. Not visible here.";
   //...
}
```

Likewise, the 'declare warning' should be changed to 'declare error' when all the warnings are gone.

Example: protecting an inter-type method relative to both the host class and the aspect chains

What if we want to allow access to a member in the host class as well as in the aspect and their descendents? Doing that with the above example simply requires one more within() to the above 'declare error':

```
declare error:
   call(void processSensitiveData())
   && !within(Participant+)
   && !within(GeneralPolicy+):
   "Call to processSensitiveData () outside Participant and GeneralPolicy chains.";
```

2.2.4 Replace Inter-type Field with Aspect Map

Typical situation

An aspect resorts to inter-type declarations to introduce additional state to a set of classes, when something more dynamic and flexible would be desirable.

Recommended action

Replace the inter-type declarations with a structure owned by the aspect and capable of mapping the target objects to the additional state.

Motivation

An inter-type declaration is a static mechanism. It affects all instances of the target class, throughout their entire life cycles. For some problems, this is exactly right, but for others something more flexible would be preferable. In some cases only a subset of all instances of a class need the extra state and behaviour, or they need it only in a specific phase of their life cycles. Sometimes the same instance simultaneously needs multiple instances of the extra state and behaviour. Sometimes the application only knows at runtime which instances need the extra state and behaviour. Inter-type declarations do not provide the necessary flexibility in these cases.

Hannemann and Kiczales showed in the code they presented in [11] how to meet these requirements in an elegant way. According to their designs, the aspect owns a data structure mapping the specific instances to the extra state. In many cases the mapping functionality can be implemented using an instance of one of Java's collections (usually a hash table), plus a few aspect methods to manage it.

An inter-type declaration is itself a kind of mapping, usually from a class to a field or method. The problem is that we cannot control the moments when it applies, when it ceases to apply, and the precise set of objects to which it applies. Whenever this kind of flexibility is required and the existing solution relies on introductions, use *Replace Inter-type Field with Aspect Map* to replace the introductions with a suitable mapping.

This refactoring is also useful in a different situation. Sometimes we have several aspects performing similar actions on similar data, and these include inter-type declarations. Naturally, we'll want to remove the duplication, by pulling the common parts to a superaspect. Here arises another problem: as long as each subaspect introduces its own additional fields to classes, there will be separate instances of the additional state for each subaspect. However, if the code is pulled up to the superaspect, there will be a single instance of the introduced state common to all subaspects. A similar problem would arise if we tried to replace an instance field with a static field. Such a pull will almost certainly not be behaviour preserving. In most cases, an intertype declaration cannot be pulled up to a superaspect as is. This pulling up usually requires the prior replacement of inter-type state with aspect state.

As it happens, the kind of replacements that solve the first problem can solve the second problem as well. This is so because there is a separate instance of the state declared in the superaspect in each active subaspect, unlike with inter-type declarations. In most cases, solving these problems is an issue of selecting a suitable structure to replace the inter-type fields and update the associated logic accordingly. To ease the replacement of the original inter-type state with the new mapping structure, it is best to first isolate it behind a small layer within the aspect, to protect the rest of the aspect code from being exposed to it. In the simplest case, all we have to do is ensure that the aspect is provided with accessor methods encapsulating the inter-type fields. Only those methods will need to be changed when the structure is replaced. In the case of preparing inter-type declarations to be pulled up, *Replace Inter-type Field with Aspect Map* must be applied to each of subaspects in turn. Next, use *Pull Up Field* ([8], p.320) and *Pull Up Method* ([8], p.322) to pull the state and its associated logic to the common superaspect.

Preconditions

When using *Replace Inter-type Field with Aspect Map* to prepare a pull to a superaspect, ensure that the fields in the various aspects do indeed provide equivalent interfaces and functionality.

Mechanics

- Use *Encapsulate Field* ([8], p.206) on the introduced field. Unlike traditional accessor methods, these ones are aspect methods, receiving the target object as argument.
- Add to each aspect a new structure capable of supporting the equivalent functionality. Add accessors similar to the ones created in the previous step: they would ideally have the same signatures and similar names. Add any additional management methods (i.e. for insertion, removal, etc.) that may also be required.
- If the aspect resorts to inter-type methods to handle the field, use *Replace Inter-type Method with Aspect Method* (33) to create aspect versions of those methods, using the new structure.
- Compile and test.
- Replace each call to the original accessors with the new ones. Compile and test when all replacements are done.
- Remove the old accessor methods. Compile and test.
- Remove the old inter-type field and related code. Compile and test.

Example: replacing an inter-type field with an aspect map

The following example presents fragments of an aspect implementing an instance of the Mediator pattern [9], adapted from a Java implementation by Cooper [5]. In this example, there is a mediator object (of type Mediator) acting as the hub of communication between various colleagues. The colleagues are instances of ClearButton and MoveButton, both subclasses of javax.swing.JButton, and KidList, which is a subclass of javax.swing.JScrollPane, implementing a listener interface from the javax.swing.event API. This example declares the Colleague role as a marker interface and assigns it to the three colleague participant types. The aspect indirectly introduces in each colleague a reference to the mediator, by way of the marker interface.

```
public aspect Mediating ...
   private interface Colleague {}
   private Mediator Colleague.mediator;

   declare parents: (ClearButton || MoveButton || KidList) implements Colleague;

   pointcut clearButtonExecute(ClearButton clearButton): ...
   after(ClearButton clearButton): clearButtonExecute(clearButton) {
      clearButton.mediator.clear();
   }

   pointcut moveButtonExecute(MoveButton moveButton): ...
   after(MoveButton moveButton): moveButtonExecute(moveButton) {
      moveButton.mediator.move();
   }

   pointcut kidListChanged(KidList kidList): ...
   after(KidList kidList) returning: kidListChanged(kidList) {
      kidList.mediator.select();
   }
```

This implementation is unsuitable because it introduces the additional state and behaviour to all instances of the participant classes, independently of whether all of them need it or not. Individual instances may never need that additional logic, may need it on only certain phases of the program execution, or may need to participate in more than one instance of the pattern at the same time. By replacing this implementation with one based on a map, we overcome all these limitations.

As a first step, we perform a refactoring similar of *Encapsulate Field* ([8], p.206) to produce a getter method for the inter-type field. The same getter can be used in all different target types. It cannot be given the same name as the final getter, so we add a zero to avoid compiler errors.

```
public aspect Mediating ...
   private Mediator getMediator0(Colleague colleague) {
      return colleague.mediator;
   }
   pointcut ...
   after(ClearButton clearButton): clearButtonExecute(clearButton) {
      getMediator0(clearButton).clear();
   }
   pointcut ...
   after(MoveButton moveButton): moveButtonExecute(moveButton) {
      getMediator0(moveButton).move();
   }
   pointcut ...
   after(KidList kidList) returning: kidListChanged(kidList) {
      getMediator0(kidList).select();
   }
```

Next, we add a suitable data structure to map the target objects to the mediator field. A hash table is a good choice for these cases. The introduced field was private to the aspect, so the getters are private as well. The access of the setter depends on the point where the mappings of the target objects to the field are made. In this example, we assume a public access.

```
import java.util.WeakHashMap;

public aspect Mediating ...
   WeakHashMap colleague2mediatorMap = new WeakHashMap();

   private Mediator getMediator(Colleague colleague) {
      return (Mediator)colleague2mediatorMap.get(colleague);
   }
   public void setMediator(Colleague colleague, Mediator mediator) {
      colleague2mediatorMap.put(colleague, mediator);
   }
```

We now add the calls to the setter in the client code. The places where the objects containing the field are created could be used as a basis, though in some cases it may be preferable to place the calls elsewhere. Outside the aspect, the calls should be something like this:

```
   Mediating.aspectOf().setMediator(clearButton, mediator);
```

Inside advice within the aspect, the same call can be expressed in a simpler way:

```
   setMediator(clearButton, mediator);
```

After we insert the calls to the setter and make the calls to the getter refer to the new getter, we can delete the original declaration and getter. Now the aspect's code looks like this:

```
public aspect Mediating ...
   private Mediator Colleague.mediator;
   declare parents: (ClearButton || MoveButton || KidList) implements Colleague;

   WeakHashMap colleague2mediatorMap = new WeakHashMap();

   private Mediator getMediator(Colleague colleague) {
      return (Mediator)colleague2mediatorMap.get(colleague);
   }
   public void setMediator(Colleague colleague, Mediator mediator) {
      colleague2mediatorMap.put(colleague, mediator);
   }

   pointcut clearButtonExecute(ClearButton clearButton): ...
   after(ClearButton clearButton): clearButtonExecute(clearButton) {
      getMediator(clearButton).clear();
   }
   pointcut moveButtonExecute(MoveButton moveButton): ...
   after(MoveButton moveButton): moveButtonExecute(moveButton) {
      getMediator(moveButton).move();
   }
```

```
    pointcut kidListChanged(KidList kidList): ...
    after(KidList kidList) returning: kidListChanged(kidList) {
        getMediator(kidList).select();
    }
```

Example: preparing an Observer implementation for the extraction of a superaspect

This second example is an implementation of the Observer pattern [9] that was extracted into an aspect from the example in [5], using *Extract Feature into Aspect* (5). This example is a bit more complex than the previous one, because it includes inter-type methods that use the inter-type field. These inter-type methods must be replaced using *Replace Inter-type Method with Aspect Method* (33). We assume the scenario in which the system has other, similar, implementations of the pattern and we would like to factor out the common elements by pulling them up to a superaspect. These implementations rely on the introduction of a java.util.Vector field to the subject participant, which is among the elements we would like to pull up, along with its associated logic.

The present implementation does not lend itself to be pulled up to the superaspect, for the same reasons as in the previous example. It was designed assuming there would be only one instance of the pattern for each subject: the vector cannot support multiple observing relationships for the same object. To solve this problem, we'll replace the inter-type vector with a more suitable hash table owned by the aspect, which will manage the mappings between subjects and the list (i.e. a java.util.Vector object) of its observers. We'll use *Replace Inter-type Method with Aspect Method* (33) to replace the logic using the vector with aspect methods using the hash table.

The example from [5] includes a Watch2LSubject object as subject and two types of observers, which are instances of ListFrameObserver and ColorFrameObserver (both subclasses of javax.swing.JFrame). The Watch2LSubject object includes three radio buttons, one for each of the colours red, green and blue. Whenever a different radio button is selected, the ColorFrameObserver instances change their background colour accordingly, and the ListFrameObserver adds the name of the selected colour to its list.

The refactored aspect uses two inner interfaces[6] to represent the roles of subject and observer. It introduces the java.util.Vector field to the objects playing the role of subject, which holds the subject's registered observers. The aspect also introduces two methods to the subjects: addObserver(Observer), which is used to register a new observer for the subject, and notifyObservers(JRadioButton), through which subjects notify all their registered observers of a change in the selected colour. That notification is carried out through the sendNotify() method, which is declared in the Observer inner interface. The sendNotify() method receives as parameter a string representing the new colour. The aspect also introduces the implementation of sendNotify() for each concrete observer type.

```
public aspect Observing ...
    private interface Subject {}
    interface Observer {
        /** notify the Observers that a change has taken place */
        public void sendNotify(String s);
    }
    declare parents: Watch2LSubject implements Subject;
    declare parents: (ListFrameObserver || ColorFrameObserver) implements Observer;

    private Vector Subject._observingFramesList = new Vector();

    public void Subject.addObserver(Observer obs) {
        //    adds observer to list in Vector
        _observingFramesList.addElement(obs);
    }
    /* sends text of selected button to all observers */
    private void Subject.notifyObservers(JRadioButton rad) {
        String sColor = rad.getText();
        for (int i = 0; i < _observingFramesList.size(); i++ ) {
            ((Observer) (_observingFramesList.elementAt(i))).sendNotify(sColor);
        }
    }
```

[6] In the initial Java implementation [5] Observer and Subject were standalone interfaces. They were inlined to within the Observing aspect during the adaptation process.

```
   public void ListFrameObserver.sendNotify(String s) {
      _listData.addElement(s);
   }
   public void ColorFrameObserver.sendNotify(String str) {
      changeColor(str);
   }
```

The aspect also includes a pointcut and corresponding advice to trigger the adequate behaviour when the subject changes the selected colour:

```
   pointcut watchStateChange(Watch2LSubject watch, ItemEvent event): ...
   after(Watch2LSubject watch, ItemEvent event): watchStateChange(watch, event) {
      if(event.getStateChange() == ItemEvent.SELECTED)
         watch.notifyObservers((JRadioButton) event.getSource());
   }
```

The mechanics prescribe the use of *Encapsulate Field* ([8], p.206) on the existing field. In this particular case, we must instead create a new field as the mapping structure (we'll create the accessor methods for the structure as soon as there is a need to do so).

```
import java.util.WeakHashMap;
...
public aspect Observing ...
   //...
   WeakHashMap _subject2Observers = new WeakHashMap();
```

Next, we use *Replace Inter-type Method with Aspect Method* (33) to replace the addObserver() and notifyObservers() inter-type methods with aspect versions using the mapping structure (see the example section of *Replace Inter-type Method with Aspect Method* (33) for more details of this step).

The new implementation is now in place and working. There was no need to add accessors to the mapping structure, as it is already encapsulated by addObserver() and notifyObservers(). These two aspect methods comprise a small layer hiding the structure. We can now delete the old implementation, after which the aspect looks like this:

```
public aspect Observing ...
   private interface Subject {}
   interface Observer {
      /** notify the Observers that a change has taken place */
      public void sendNotify(String s);
   }
   declare parents: Watch2LSubject implements Subject;
   declare parents: (ListFrameObserver || ColorFrameObserver) implements Observer;

   WeakHashMap _subject2Observers = new WeakHashMap();

   public void addObserver(Subject subject, Observer observer) {
      Vector observers;
      Object obj = _subject2Observers.get(subject);
      if(obj == null)
         observers = new Vector();
      else observers = (Vector) obj;
      observers.add(observer);
      _subject2Observers.put(subject, observers);
   }
   public void notifyObservers(Subject subject, JRadioButton radioButton) {
      String sColor = radioButton.getText();
      Vector observersList = (Vector)_subject2Observers.get(subject);
      for (int i = 0; i < observersList.size(); i++ ) {
         ((Observer) (observersList.elementAt(i))).sendNotify(sColor);
      }
   }
   public void ListFrameObserver.sendNotify(String s) {
      _listData.addElement(s);
   }
   /* Observer is notified of change here */
   public void ColorFrameObserver.sendNotify(String str) {
      changeColor(str);
   }

   pointcut watchStateChange(Watch2LSubject watch, ItemEvent event): ...
   after(Watch2LSubject watch, ItemEvent event):
         watchStateChange(watch, event) {
      if(event.getStateChange() == ItemEvent.SELECTED)
         notifyObservers(watch, (JRadioButton) event.getSource());
   }
}
```

2.2.5 Replace Inter-type Method with Aspect Method

Typical situation

An aspect resorts to inter-type declarations to introduce additional methods to a class or interface, when a more dynamic and flexible composition mechanism is required.

Recommended action

Replace the inter-type method with a method owned by the aspect.

Motivation

In most cases, a method introduced to a class can be replaced by a similar aspect method receiving an instance of the target class as an additional argument. This provides the basis for this refactoring.

```
public class Capsule {
   private int _value;
   public Capsule(int value) {
      _value = value;
   }
```
```
public aspect Additional {
   public void Capsule.doSomethingMore() {
      System.out.println("Doing something more with capsule" + this);
   }
```
```
      Capsule capsule = new Capsule(7);
      capsule.doSomethingMore();
```

↓

```
public class Capsule {
   private int _value;
   public Capsule(int value) {
      _value = value;
   }
```
```
public aspect Additional {
   public void doSomethingMore(Capsule capsule) {
      System.out.println("Doing something more with " + capsule);
   }
```
```
      Capsule capsule = new Capsule(7);
      Additional.aspectOf().doSomethingMore(capsule);
```

Replacements of this kind should not be made in the general case. In some particular cases, it may be a desirable thing. The motivation for doing this is the same as the one stated for *Replace Inter-type Field with Aspect Map* (28). When the additional state and behaviour introduced by an aspect is needed by only a subset of the instances, or they need it only during certain phases, or they need multiple instances of that state and behaviour, the mechanism of inter-type declarations is not flexible enough. In addition, the use of this refactoring is also useful when using *Replace Inter-type Field with Aspect Map* (28) in cases in which the inter-type field being replaced is used by existing inter-type methods. In such cases, *Replace Inter-type Method with Aspect Method* can be used to create the methods using the map that results from using *Replace Inter-type Field with Aspect Map* (28).

Mechanics

- Create in the aspect a copy of the inter-type method, with same name and signature. Insert, in the beginning of the aspect method's parameter list, an additional parameter whose type is the original target of the inter-type declaration.
- Replace each reference to 'this' with the new parameter. Change all self-calls and references to fields to refer to the first parameter.
- Compile and test.
- Change the body of the inter-type method so that it calls the aspect method, if it can be done at this point.
- Add a 'declare warning' exposing all calls to the inter-type method:

```
declare warning:
     (call(<type> <host class>.someMethod(<arguments>)):
     "method <host class>.someMethod() is called here.";
```

- Following the warnings, replace each call to the inter-type method with a call to the aspect method. Compile and test after each change.
- When there are no more warnings remove the 'declare warning' and the inter-type method.

  When covering the mechanics of several refactorings from [8] Fowler considers the situation when the existing method is part of the interface and cannot be changed. He recommends that in such cases the old method be left in place and marked as deprecated.
- Compile and test.

Example

This example is part of the second example for *Replace Inter-type Field with Aspect Map* (28). In it, an aspect introduces the following methods to the Subject marker interface:

```
public void Subject.addObserver(Observer obs) {
   _observingFramesList.addElement(obs);
}
private void Subject.notifyObservers(JRadioButton rad) {
   String sColor = rad.getText();
   for (int i = 0; i < _observingFramesList.size(); i++ ) {
      ((Observer) (_observingFramesList.elementAt(i))).sendNotify(sColor);
   }
}
```

As an example of client code, the following subject and observers are created and registered, through calls to Subject.addObserver():

```
   Watch2LSubject subject = new Watch2LSubject();
   //Observing.aspectOf().setSubject(subject);

   ColorFrameObserver cframeObs1 = new ColorFrameObserver();
   ColorFrameObserver cframeObs2 = new ColorFrameObserver();
   ColorFrameObserver cframeObs3 = new ColorFrameObserver();
   ListFrameObserver lframeObs = new ListFrameObserver();

   subject.addObserver(cframeObs1);
   subject.addObserver(cframeObs2);
   subject.addObserver(cframeObs3);
   subject.addObserver(lframeObs);
```

The aspect itself also includes an advice calling the other method, Subject.notifyObservers():

```
after(Watch2LSubject watch, ItemEvent event):
      watchStateChange(watch, event) {
   if(event.getStateChange() == ItemEvent.SELECTED)
      watch.notifyObservers((JRadioButton) event.getSource());
}
```

This functionality is to be replaced by aspect methods using a hash table owned by the aspect – aspect field _subject2Observers, which contains subjects as keys, and vectors of observers as values:

```
WeakHashMap _subject2Observers = new WeakHashMap();
```

As a first step, we create the following two aspect methods, with the same names:

```
public void addObserver(Subject subject, Observer observer) {
   Vector observers;
   Object obj = _subject2Observers.get(subject);
   if(obj == null)
      observers = new Vector();
   else observers = (Vector) obj;
   observers.add(observer);
   _subject2Observers.put(subject, observers);
}
public void notifyObservers(Subject subject, JRadioButton radioButton) {
   String sColor = radioButton.getText();
   Vector observersList = (Vector)_subject2Observers.get(subject);
   for (int i = 0; i < observersList.size(); i++ ) {
      ((Observer) (observersList.elementAt(i))).sendNotify(sColor);
   }
}
```

We can’t replace the body of the inter-type methods with calls to the new ones at this point. We must first replace the calls to addObserver(), which register the observers to their subjects. Otherwise the tests wouldn’t pass. We therefore perform the next step as prescribed, adding ‘declare warning’ clauses that will expose all calls to the these methods:

```
declare warning: call(void Subject.addObserver(Observer)):
   "Method Subject.addObserver(Observer) is called here.";
declare warning: call(void Subject.notifyObservers(JRadioButton)):
   "Method Subject.notifyObservers(JRadioButton) is called here.";
```

We compile, resulting in a series of warnings locating the calls to the old methods. After replacing each of them with calls to the aspect methods, we compile again. All warnings disappeared, and we test. We remove the ‘declare warning’ clauses. Now the client code calling addObservers() looks like this:

```
Watch2LSubject watch2LFrame = new Watch2LSubject();

ColorFrameObserver cframeObs1 = new ColorFrameObserver();
ColorFrameObserver cframeObs2 = new ColorFrameObserver();
ColorFrameObserver cframeObs3 = new ColorFrameObserver();
ListFrameObserver lframeObs = new ListFrameObserver();

Observing.aspectOf().addObserver(watch2LFrame, cframeObs1);
Observing.aspectOf().addObserver(watch2LFrame, cframeObs2);
Observing.aspectOf().addObserver(watch2LFrame, cframeObs3);
Observing.aspectOf().addObserver(watch2LFrame, lframeObs);
```

And the call to notifyObservers() now takes the form:

```
after(Watch2LSubject watch, ItemEvent event):
      watchStateChange(watch, event) {
   if(event.getStateChange() == ItemEvent.SELECTED)
      notifyObservers(watch, (JRadioButton) event.getSource());
}
```

2.2.6 Tidy Up Internal Aspect Structure

Typical situation

You've just extracted the various elements of the implementation of a crosscutting concern into a new aspect but its internal structure turns out to be sub-optimal.

Recommended action

Tidy up the internal structure of the new aspect to eliminate duplication.

Motivation

AOP adds a new type of situation in which code duplication can arise. Refactoring an object-oriented (OO) code base to aspects entails extracting crosscutting concerns and features to aspects. The very crosscutting nature of the extracted concerns makes duplication very likely to occur in the various classes touched by the concern. In many situations, this duplication is a direct consequence of limitations in the composition mechanisms of traditional OO systems. For instance, a system may contain repeated implementations of the same functionality scattered in multiple classes because previously these could not be modularised.

Simply extracting those code snippets into an aspect does not guarantee, by itself, that the duplication is removed. It merely moves the duplicated code into aspects. In some cases, the duplication only may become obvious when it is placed in a single module. Therefore extracting the code related to crosscutting concerns into aspects is only the first part of the job. Next, there is the task of removing duplication within the aspect and improving its internal structure. The necessary refactorings may involve profound changes in the structure of the aspects. In some cases, the result may be the replacement of the extracted design with a different, more suitable design.

Another problem can arise with aspects resulting from extractions of crosscutting concerns, related to inter-type declarations. They make it very easy to move members from classes to aspects without impact on client code, and any aspect resulting from extractions is very likely to include them.

However, in some cases, we would like the aspect to introduce the additional state and behaviour on an object-by-object basis, and inter-type declarations are not flexible enough to achieve that. This entails the replacement of these introductions with different logic, to keep still within the aspect.

Mechanics

- If the code assigns roles to participant classes, ensure that marker interfaces are used to represent those roles rather than referring directly to concrete classes. If it is not the case, use *Generalise Target Type with Marker Interface* (24).
- If parts of the code make explicit references to specific classes that cannot be generalised, separate the specific parts from the generally applicable ones by using *Extract Method* ([8], p.110).
- Inspect the inter-type declarations looking for cases in which the objects only need the extra state and behaviour at specific times, or may need more than one instance of it at a single time, or only a subset of the instances of an introduced class actually needs it. In such cases, consider using *Replace Inter-type Field with Aspect Map* (28) to replace introduced fields with state directly managed by the aspect, and *Replace Inter-type Method with Aspect Method* (33) to perform a similar transformation in respect of introduced methods.

2.3. Dealing with Generalisation

The main refactoring for this group is *Extract Superaspect* (36). This group bears the same name as an equivalent group in [8], which contains various *Pull Up* and *Push Down* refactorings. This section contains similar refactorings dealing in this case with aspect-specific constructs, including pointcuts, advice, marker interfaces and inter-type declarations.

2.3.1 Extract Superaspect

Typical situation

You have two or more aspects containing similar code and functionality.

Recommended action

Create a superaspect and move the common features to the superaspect.

Motivation

It has been noted numerous times that common patterns sometimes only surface during development. When a common pattern in the code is identified, the sensible thing to do is make that commonality clear, by creating a separate unit of modularity with an appropriate name and placing the common code there. An obvious benefit is the removal of duplication.

When we extract various concerns from a code base, we may later conclude that several of the resulting aspects are in fact different instances of a common pattern. Different aspects may turn out to be variant implementations of the same kind of functionality, but sometimes the similarities only become noticeable when we are able to see all the code of each concern in one place (commonalities among aspects should also be easier to notice after applying *Tidy up Internal Aspect Structure* (35) to each of them).

When such commonalities are identified, it is necessary to prepare the internal structure of the aspects so that the common parts are amenable to being extracted into a common superaspect. In the case of inter-type declarations, the needed action usually entails the replacement of the existing introductions with a different implementation, as suggested in *Replace Inter-type Field with Aspect Map* (28) and *Replace Inter-type Method with Aspect Method* (33).

Mechanics

- Create an empty superaspect. Make the concrete aspects inherit from the abstract aspect.
- One by one, use *Pull Up Marker Interface* (39), *Pull Up Field* ([8], p.320), *Pull Up Method* ([8], p.322), *Pull Up Pointcut* (39) and *Pull Up Advice* (37) to move common elements to the superaspect.

  It's usually easier to move the marker interfaces first.

  Pull Up Field ([8], p.320) and *Pull Up Method* ([8], p.322) cannot be used with inter-type declarations, because there will be only one instance of the introduced fields for all subaspects. When you want to pull up inter-type fields, consider applying *Replace Inter-type Field with Aspect Map* (28) to the inter-type fields and *Replace Inter-type Method with Aspect Method* (33) to the inter-type methods using those fields.

  If you find a signature common to all subaspects but with different logic, add in the superaspect an abstract declaration or default definition.
- Compile and test after each pull.

2.3.2 Pull Up Advice

Typical situation

Several subaspects use the same advice acting on a pointcut declared in the superaspect.

Recommended action

Move the advice to the superaspect.

Motivation

As with other kinds of pulls, this situation is likely to arise when a commonality is being extracted from various subaspects.

At first sight, it may look that most of the considerations applying to *Pull Up Method* also apply here, but there are important differences. First, there is no overriding: all pieces of advice defined along the inheritance chain execute whenever one joinpoint is reached. There is no polymorphism. In addition, keep in mind that a piece of advice declared in a superaspect will run as many times as there are concrete subaspects weaved into the system. If that is not what you want, leave the advice in the subaspects. This won't be a problem if at any given time only one concrete subaspect is weaved into the system.

Preconditions

If the common pointcut over which the advice executes is still duplicated in the various subaspects, use *Pull Up Pointcut* (39) first.

If at least one of the subaspects use a piece of advice different from the others you should not pull up the advice, since pieces of advice cannot be overridden.

Mechanics

- Copy the body of the advice in the superaspect.
- Delete the advice in each of the subaspects.
- Compile and test.

2.3.3 Pull Up Declare Parents

Typical situation

Two concrete aspects use the same ‘declare parents’.

Recommended action

Move the 'declare parents' to the superaspect.

Motivation

This refactoring contributes to extract a reusable superaspect out of two or more concrete subaspects that duplicate some logic.

Mechanics

- Ensure that all 'declare parents' assign the same roles to the same participants.
- Place a copy of the 'declare parents' in the superaspect.
- Delete the 'declare parents' in each of the subaspects.
- Compile and test.

2.3.4 Pull Up Inter-type Declaration

Typical situation

An inter-type declaration would be best placed in the superaspect than where it presently stands.

Recommended action

Move the inter-type declaration to the superaspect.

Motivation

The pull up or push down of inter-type declarations presents issues not generally found in other aspect constructs. For this reason, the applicability of this refactoring is considerably more restricted than for other *Pull Up* refactorings. The main motivation of other *Pulls* is the factoring out of commonality out of duplicated code. Sometimes it is also convenient to move an element to the superaspect even when there is no duplication, because we simply think it is best placed there. This refactoring applies to only this last case. This main reason is the fact that the number of instances of the introduced member is different depending of where the inter-type declaration is placed in the aspect inheritance chain. Target objects (i.e. instances of classes affected by the inter-type declaration) will have one separate instance of the inter-type member for each subaspect. If the various inter-type declarations are factored out to a single declaration in a superaspect, the target objects will have just one instance of the introduced member. This situation is roughly similar to when an instance member is turned into a static member. Such a transformation is not likely to be behaviour preserving, particularly when applied to fields. Also, keep in mind that two or more inter-type declarations of the same member will conflict with each other if their scopes of visibility overlap. Consequently, if several subaspects declare the same member, those members are likely to be private to the subaspects (this would present an additional hurdle, if the mechanics entailed first making them public).

Preconditions

In the general case, this refactoring can be used only when there is a single instance of the inter-type declaration to pull up. Otherwise, whenever inter-type members include both fields and methods, deal with the fields first, using *Replace Inter-type Field with Aspect Map* (28) (the inter-type methods may themselves be refactored during that process). *Pull Up Inter-type Declaration* can also be applied to duplicated inter-type methods that do not refer to inter-type fields.

Mechanics

- Create a new inter-type declaration in the superclass. If the declaration is private, relax the access to public and use *Introduce Aspect Protection* (26).
- Delete the inter-type declaration in the subaspect.

- Compile and test.

2.3.5 Pull Up Marker Interface

Typical situation

Two concrete aspects use a marker interface to model the same role.

Recommended action

Move the marker interfaces to the superaspect.

Motivation

This refactoring contributes to extract a reusable superaspect out of two or more concrete subaspects that duplicate some logic.

Preconditions

Marker interfaces do not generally need to declare signatures. If the interface is not blank, consider first refactoring the code in order to remove the signatures.

Mechanics

- Inspect all uses of the interfaces to ensure the roles they model are indeed the same. Ensure they have the same name.
- Create a new interface in the superaspect. If the interfaces are *private,* you will need to change them to *protected*.
- Delete the subaspect interfaces.
- Compile and test.

Example

```
public abstract aspect AbstractMediation {
   //...
}
```

```
public aspect MediationCase1 extends AbstractMediation {
   private interface Mediator { }
   //...
}
```

```
public aspect MediationCase2 extends AbstractMediation {
   private interface Mediator { }
   //...
}
```

↓

```
public abstract aspect AbstractMediation {
   protected interface Mediator { }
   //...
}
```

```
public aspect MediationCase1 extends AbstractMediation {
   //...
}
```

```
public aspect MediationCase2 extends AbstractMediation {
   //...
}
```

2.3.6 Pull Up Pointcut

Typical situation

Two or more aspects declare identical pointcuts.

Recommended action

Move the pointcut to the superaspect.

Motivation

The mechanics of this refactoring have strong similarities with that of *Pull Up Method* ([8], p.322).

Mechanics

- Inspect the pointcuts to ensure they are identical. If the pointcuts are similar but not identical, first see if both capture the same set of joinpoints. Next, change one of them into the other. If the pointcuts have different signatures, change the signatures to the one you intend to use in the superaspect.

  Often, the pointcuts relate to the same concept and have the same signature, but capture different sets of joinpoints in each specific case. In such cases keep the pointcuts in the subaspects and place an abstract declaration in the superaspect.

- Create a copy of the new pointcut in the superaspect. If the pointcuts are private or package protected classify the new pointcut as protected. If the pointcut uses other pointcuts that are present in the aspects but not in the superaspect, declare the corresponding abstract pointcuts on the superaspect.
- Delete the pointcuts in the subaspects.
- Compile and test.

2.3.7 Push Down Advice

Typical situation

A piece of advice is used only by some subaspects, or each subaspect requires a different advice.

Recommended action

Move the advice to the subaspects that use it.

Motivation

This situation can arise when we discover that a supposedly reusable piece of advice does not cover all cases after all. Since there is no polymorphism with advice, you cannot override the advice with a case-specific advice in the subaspect. In these cases it is necessary to place a version of the advice in each subaspect.

Mechanics

- Copy the advice to each subaspect. Check for any imports that the subaspects may require resolving the advice's code.
- Remove the advice from the superaspect. Check for any imports that may no longer be needed.
- Compile and test.

Example

The following advice comes from ObserverProtocol, the reusable aspect for the Observer pattern by [11]:

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

Suppose you want to implement a use case similar to Eckel's flower example of the same pattern [6], in which observers are notified only the first time a distinct event occurs, avoiding repeated notifications when the same event occurs without other events in between. We copy the advice to the subaspect and remove it from ObserverProtocol:

```
public abstract aspect ObserverProtocol {
    //...
```

```
public aspect ObservingOpen extends ObserverProtocol {
    //...
    after(Subject s): subjectChange(s) {
        Iterator iter = getObservers(s).iterator();
        while ( iter.hasNext() ) {
            updateObserver(s, ((Observer)iter.next()));
        }
    }
```

You can then add the intended functionality.

Naturally, if you have several other subaspects of ObserverProtocol that are quite happy to use the former advice, it would be bad style to duplicate the advice in each of them. In such cases, it is preferable to parameterise the additional behaviour in the advice.

2.3.8 Push Down Declare Parents

Typical situation

A 'declare parents' in a superaspect is not relevant for all its subaspects.

Recommended action

Move the 'declare parents' to the subaspects where it is relevant.

Motivation

Push Down Declare Parents is the opposite of *Pull Up Declare Parents* (37). Declare parent clauses are more likely to be found in case-specific concrete subaspects than in abstract superaspects. The concrete aspects are used to inherit some reusable logic from an abstract aspect, and the latter does not generally includes 'declare parents' clauses.

Mechanics

- Add the 'declare parents' in all subaspects that require it. Compile and test.
- Remove the 'declare parents' from the superaspect. Compile and test.

2.3.9 Push Down Inter-type Declaration

Typical situation

An inter-type declaration would be best placed in a subaspect.

Recommended action

Move the inter-type declaration to the subaspect.

Motivation

The pull up or push down of inter-type declarations presents issues not generally found in other aspect constructs. It is possible to declare the same member in multiple aspects, but their visibility scopes cannot overlap. This generally means they must be private. In addition, pushing an inter-type declaration down entails replacing a single instance (per target object) of the member common to all subaspects with one instance (per target object) of the inter-type member per subaspect. When the member is a field, this is sure not to be behaviour preserving, just as with the opposite refactoring, *Pull Up Inter-type Declaration* (38). When the member is a method, this leads to duplication.

Preconditions

For the above reasons, this refactoring should be used to push down the inter-type declaration to only one subaspect.

Mechanics

- Copy the declaration in the subaspect.
- Remove the declaration from the superaspect.
- Compile and test.

2.3.10 Push Down Marker Interface

Typical situation

A marker interface declared within a superaspect models a role that is used only in some of its superaspects.

Recommended action

Move the marker interface to those subaspects.

Motivation

Push Down Marker Interface is the opposite of *Pull Up Marker Interface* (39), and can be used to reverse its effects when they do not turn out as expected.

Preconditions

Inspect the superaspect to ensure it does not include any references to the marker interface, besides the declaration itself. If the interface is not private, also ensure no client code refers to the interface as belonging to the superaspect.

Mechanics

- Declare the marker interface in each of the superaspects.
- Delete the declaration from the superaspect.
- Compile and test.

2.3.11 Push Down Pointcut

Typical situation

A pointcut is not used by some subaspects.

Recommended action

Move the pointcut to those subaspects.

Motivation

A pointcut represents a set of interesting events that should trigger some response on the part of the aspects. Placing the pointcut in an abstract aspect leads programmers to assume that those events are interesting to any descendent of the abstract aspect. When that is not so, it can lead to confusion. If some of the aspects do not need the pointcut at all, this is an instance of *Refused Bequest* ([8], p.87) applied to pointcuts. It is advisable to analyse the specific case to be sure this smell is worth cleaning, as with more traditional instances of this smell.

Different cases to be considered as well are the ones in which only the concrete aspects know what are the interesting events. In this case, you should leave a protected abstract declaration of the pointcut in the superaspect. In addition, there are cases when the pointcut to be pushed down is used by advice or other pointcuts in the superaspect. These cases also require an abstract declaration of the pointcut.

Keep in mind that any inherited abstract declaration must be concretised in all concrete subaspects. If you really want to refuse the bequest you can override the inherited pointcut with one that doesn't captures any joinpoints, such as in the example below. However, this solution does not remove the *Refused Bequest* smell and it would be best to use it only as a stopgap.

```
public abstract AbstractAspect {
   protected abstract pointcut refusedBequest(<parameters>);
```

```
public ConcreteAspect extends AbstractAspect {
   protected pointcut refusedBequest(<parameters>): !within(*);
```

Mechanics

- Declare the pointcut in all subaspects.
- Remove the pointcut from the superaspect. Leave an abstract declaration if one of the cases requiring one applies.
- Compile and test.
- If you did not leave an abstract declaration in the superaspect, remove the pointcut from any subaspect that does not need it.

2.4. Dealing with Legacy Code

The refactoring covered in this section was specifically designed for dealing with legacy code with published interfaces[7]. By published interfaces, we mean APIs that are used by clients outside the control of the developer, meaning that he is unable or not authorised to change.

2.4.1 Partition Constructor Signature

Typical situation

You're extracting from the primary code to an aspect all the code related to a particular concern. A constructor in the primary code has initialisation code that uses values coming from some of the constructor's arguments. These arguments are not required when the primary code does not include the extracted concern.

Recommended action

Create in the class a constructor devoid of any code relative to the extracted concern, including the arguments. Replace the code from original constructor that is not related to the extracted concern with a call to the new constructor. Move the original constructor to the aspect.

Mechanics

This awkward situation arises when an interface is tangled with various concerns but for some reason cannot be changed, or we want to avoid our refactorings to impact on client code.

Mechanics

- Create a new constructor in the class, with a shortened argument list, without the arguments related to the crosscutting concern.
- Move to the new constructor all the statements not related to the crosscutting concern.
- Place a call to this() as the first statement in the original constructor, passing only the parameters not related to the crosscutting concern.

[7] This refactoring stemmed from the case study described in [14]. The line of research dealing with the specific problems of legacy code was not developed further.

- Move the original, modified constructor to the aspect.
- Append '.new' after the original constructor's name in the aspect. For instance, suppose arg1 is not related to the crosscutting concern:

```
//In the aspect
public
SomeClass.new(Type1 arg1, Type2 arg2) {
     this(arg1);
}
```

- In case the aspect is placed in a separate package, check if the constructor's class is declared in the aspect's import section. Check also if all imports in the host class are still necessary: some may have been needed only for arguments moved to the aspect.
- Compile and test.

Example

```
public class TangledStack {
   //...
   public TangledStack(JFrame frame) {
      elements = new Object[S_SIZE];
      frame.getContentPane().add(_label);
      text.setText("[]");
      frame.getContentPane().add(_text);
   }
   //...
}
```

↓

```
public class TangledStack {
   //...
   public TangledStack() {
      elements = new Object[S_SIZE];
   }
   //...
}
```

```
public aspect WindowView {
   //...
   public TangledStack.new(JFrame frame) {
      this();
      frame.getContentPane().add(_label);
      _text.setText("[]");
      frame.getContentPane().add(_text);
   }
   //...
}
```

For a more complete example using this refactoring, see the Example section of *Extract Feature into Aspect* (5).

3. CONCLUSION

This report presents a catalogue of 28 refactorings specific to the AspectJ programming language. The refactorings are presented in a style and format similar to the one used in the book by Fowler [8]. The catalogue is structured in the following groups:

- 10 refactorings for the extraction of crosscutting concerns from Java code bases to aspects.
- 6 refactorings for improving the internals of aspects. Several of these refactorings were specifically designed to restructure the internal structure of aspects stemming from extraction processes performed according to refactorings of the previous group.
- 11 refactorings to deal with generalisation. The refactorings in this group deal primarily with the extraction of common code between multiple aspects and the associated movement of aspect-specific constructs between superaspects and subaspects.
- One refactoring specifically designed to deal with a special case that can arise in legacy code. Its purpose is to ease the separation of concerns in constructors that are part of published interfaces.

4. REFERENCES

[1] AHEAD tool suite. http://www.cs.utexas.edu/users/schwartz/ATS.html

[2] Homepage of the WorkSCo project. http://www.esw.inesc-id.pt/worksco/

[3] K. Beck. Extreme Programming Explained: Embrace Change. Addison-Wesley 2000. ISBN 0201616416.

[4] G. Bracha, W. Cook “Mixin-Based Inheritance”, Proceedings of ECOOP/OOPSLA 1990, 303-311.

[5] J. Cooper. Java Design Patterns: A Tutorial. Addison-Wesley 2000. ISBN: 0201485397. Also availabe at http://www.patterndepot.com/put/8/DesignJavaPDF.ZIP or http://www.patterndepot.com/put/8/DesignJava.PDF.

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

[7] S. Fernandes, J. Cachopo, A. R. Silva, Supporting Evolution in Workflow Definition Languages, SOFSEM 2004, January 2004.

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

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

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

[11] J. Hannemann and G. Kiczales. Design pattern implementations in Java and AspectJ. Proceedings of the OOPSLA 2002, pages 161–173.

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

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

[14] M. Monteiro and J. Fernandes. Object-to-Aspect Refactorings for Feature Extraction. Industry paper presented at the industry tack at AOSD 2004, March 2004. Available at http://aosd.net/2004/archive/Monteiro.pdf.

[15] M. Monteiro, J. Fernandes, Refactoring a Java Code Base to AspectJ – An Ilustrative Example, practitioner’s report submitted to the AOSD'2005.