Object-to-Aspect Refactorings for Feature Extraction

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

João Miguel Fernandes
Departamento de Informática
Universidade do Minho
Campus de Gualtar
4710-057 Braga PORTUGAL
jmf@di.uminho.pt

January 2004

ABSTRACT

This report describes an experiment in using AspectJ to extract a feature from a Java code base in order to make it unpluggable. We describe issues and obstacles encountered while performing a series of code transformations and next present a collection of manual aspect-oriented refactorings, based on the experience gained in the process. These are described in detail and compounded with a self-contained example placing each refactoring in its proper context.

1. INTRODUCTION

Aspect-oriented programming (AOP) [11] and refactoring both attempt to cope with the permanent need for evolution of present-day software. The approach of refactoring aims to facilitate the continuous change of source code required for software to continuously adapt to changing environments and requirements. AOP provides stronger modularisation and software composition mechanisms than traditional technologies, thus promising to diminish the impact that changes to the code related to a given concern have on non-related code.

There is the prospect in the near future of a widespread adoption of the concepts from aspect-orientation, which begs the question of how to deal with a large legacy of OO code. Currently there is no AOP equivalent to the catalogue of object-oriented refactorings presented in [6]. We aim to fill that gap and this report presents our first results, using AspectJ [10] as the AOP language. We present the refactorings in such a way to be directly useful to programmers in front of the keyboard, editing code. Pertinent issues include the right order of code transformations, their mechanics, which transformations are adequate for certain typical situations, which preconditions should be required for each case, and we pinpoint situations in which the structure of the legacy code may influence the choice of the next transformation.

We do not aim to cover techniques for automatic support for refactoring operations [16], instead we identify and characterise the operations when performed manually. We recognise that automatic support for object-to-aspect refactorings is the ultimate goal, but we believe prior knowledge about the nature and mechanics of the transformations is beneficial. In order to discover interesting refactorings we took a non-trivial code base as a case study and performed a refactoring experiment to gain useful insights. Here we focus on the extraction of concerns that cannot be unplugged from the primary code because scattered across multiple methods and classes. The refactorings presented in this report stem from that experiment.

The rest of this report is organised as follows. In section 2 we describe the WorkSCo framework, the subject of our first refactoring experiment. In section 3 we describe it, mentioning issues and obstacles encountered and our solutions to them. In section 4 we present a collection of refactorings based on the experience we gained. This presentation is completed with a self-contained example presented in section 5. In section 6 we draw some conclusions, briefly survey related work and propose future work. In section 7 we conclude the report.

2. CASE STUDY

WorkSCo (Workflow with Separation of Concerns) is an object-oriented framework for workflow management systems currently being developed by the ESW [2] group at INESC-ID, Portugal. By "object-oriented" (OO) we mean that WorkSCo was developed using "traditional" technology including design patterns and component concepts. Until now WorkSCo does not rely on the infra-structure for specific business environments such as J2EE. Some of the functionality provided by these environments (e.g. persistence) is currently being developed as extension modules.

Having a close contact with the WorkSCo team was a strong point in favour of selecting it as a case study. Another was that WorkSCo 's developers are aware of AOP developments and have a grasp of related concepts, facilitating the exchange of ideas.

The architecture of WorkSCo was based on micro-workflow, the architecture developed by Manolescu in his Ph.D. thesis [14], using Smalltalk technology. Manolescu's architecture was the first to target developers of OO software rather than end-users. The design of micro-workflow relies on a considerable number of design patterns, most of which are intended to achieve particular separations of concerns. It comprises a lightweight kernel providing basic workflow functionalities that are compounded by extension modules offering more advanced features. Software developers select the features for their domain-specific workflow processes and add the corresponding modules through composition.

One of the kernel's key abstractions is the concept of *procedure* (Figure 1), which models various kinds of activities commonly performed by workflow systems. The various types of procedures are organised according to the Composite design pattern [6] to abstract the concrete types of steps or activities comprising the workflow definition, thus allowing the easy extension of the framework with new procedure types. These include both atomic steps (instances of simple procedure) and composite activities (e.g. composite procedure). Examples of atomic procedures are *primitive procedure*, which model the actions performed by a single domain-object, and *work list procedure*, which is an abstraction of a piece of work carried out by a human element of the organisation. Composite procedures, namely *SequenceProcedure*, *ConditionalProcedure* and *Repeat-UntilProcedure*, model the conditional and iterative control structures similar to those found in many programming languages. They include steps (child procedures) that are also procedures (either simple or composite). Thus a workflow definition has a tree structure, in which all leaf nodes must be instances of *Simple-Procedure* (e.g. primitive procedure or work list procedure) and all non-leaf nodes must be instances of *CompositeProcedure*.

The procedure system provides the basic framework for workflow management, enabling domain objects, which typically change less frequently than the glue code that models processes, to be reused as process rules evolve and different process instances are created. Micro-workflow is capable of holding various workflow definitions (representations of workflow processes) using a structure that follows the *Type Object* pattern [12].

The ESW group adapted the original micro-workflow design for their project, which is based on Java technology. In the process they introduced several structural innovations in order to attain further separations. The primary design change was motivated by the lack of a standard for workflow definition languages (WfDLs). This made it highly desirable that WorkSCo be able to support several differing (and evolving) WfDLs, and also to support more than one WfDL simultaneously. The adopted solution was to split the design between two layers – a front-end and a back-end [4]. The back-end layer comprises a *Workflow engine*, which accepts and runs low-level graph representations of the workflow definitions. Each instance of the front-end layer is responsible to handle the concepts and abstractions of a specific WfDL and to generate the graph structure the back-end understands. The WfDL currently supported by WorkSCo is micro-workflow, which thus became the first instance of the front-end layer.

Each front-end instance is responsible for generating the low-level graph structure for the back-end. In the case of micro-workflow it is done though a traversal of the workflow definition's procedure tree. The graph generation is done bottom-up, starting with the leaf nodes and navigates upwards until the root node is reached. Each node provides a compile operation that generates the graph representation of the sub-tree from which that node is the root. Non-leaf nodes (composite procedures) are responsible to bind the sub-graphs generated by their children, so that the compile operation of root node produces the graph representation of the entire workflow definition.

2.1. The data link concern

Although WorkSCo's front-end model is control-driven it enables the independent modelling of data flow between procedures, which is done through *data links*. These provide the necessary framework for communication and data passing among independent domain-nodes. They ensure the independence of the workflow steps and their reusability, and also enable the system to provide various services, for example logging and monitoring. The concept of data link belongs to the front-end, while the back-end uses a different, lower level, representation.

Data links are defined at the composite procedure level, so that all control structures have access to it. However front-end rules must still be enforced, to ensure data link specifications are consistent with the control flow. For instance, data links can only be placed in composite procedures specifying some ordering of their children (e.g. it wouldn't make sense to establish a link between the branches of a conditional procedure).

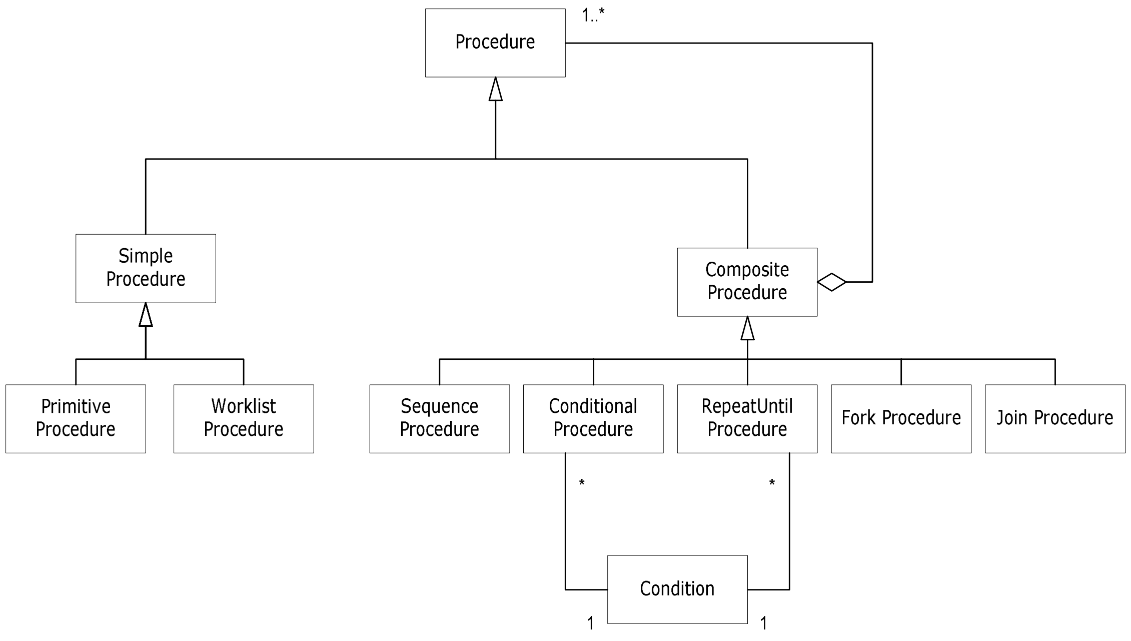

Figure 1 – The WorkSCo frontend procedure hierarchy

3. REFACTORING EXPERIMENT

The initial phase of the work, after acquiring a grasp of the important domain concepts, was to explore the code in search for units and fragments related to the target concern. Concern mining was not our main focus and we did not fully explore sophisticated mining techniques. We relied on the feedback of the WorkSCo team, which was helpful in pinpointing the data links as a feature suitable for extraction, as well as gaining the understanding of details necessary for writing unit tests.

At the time we took a snapshot of WorkSCo's code the kernel's functionality was stable and several extension modules were in various stages of development. The kernel classes and interfaces were deployed in a package containing 40 classes and interfaces plus a few subpackages providing some accessory functionality. The kernel presented very few dependencies on the code of the extension modules, the only exception being a case in which one of the key classes in the kernel included in its *implements* clause an interface declared in one of the modules. This low coupling, together with the fact that extracting the data links into an aspect affected only the kernel, enabled us to largely ignore the extension modules[1] and concentrate our analysis on the kernel.

Though generally well structured, the snapshot of the code we worked on placed a few obstacles. These were mainly (1) the fact that it did not fully adhere to the style advocated in [6], (2) the fact that the elements of both the front-end and the back-end were still bundled together in the same package, which had some impact on the initial exploration stage, and (3) the lack of unit tests. Style problems were most noticeable in the very large methods responsible for generating the back-end graph representation of workflows (the largest had 110 lines of code) and long parameter lists in constructors.

The project included a collection of various broader-scoped tests that fed the system with example workflows. Unfortunately these were unsuitable for our task, due to their coarse grain. As pointed out in [6], fine-grained unit tests are crucial to ensure the code transformations always preserve the original behaviour. From our experience we can attest the vital importance of unit tests to give what Kent Beck called "courage" in changing the existing code [3]. Only after the initial effort of writing tests did we acquire the confidence necessary to start refactoring the code (before we reached that point we simply felt "paralysed").

We used the eclipse's JDT environment [1] and the FEAT plug-in [17] to assist in our analysis. JDT's structure view and search capabilities were be very useful during the process of searching the code for units and fragments related to the target concern. Although we consider FEAT a useful complement to the code search capabilities of JDT, it was of limited use in our particular case. That was due to not covering internal details of methods such as local variables. In other words, FEAT does not capture the use relation between components, a widely used relationship in WorkSCo. WorkSCo relies less on structural relationships (i.e. inheritance) to connect components than more "traditional" OO frameworks such as the original micro-workflow [14].

[1] Another reason for ignoring the extension modules was the fact that most of them were still under active development and their functionalities still incomplete.

We wanted to preserve the existing interface of WorkSCo's kernel during refactoring. We believe this emulates a situation that occurs frequently in software projects, when there is client code that depends on existing interfaces but is out of control of the developers of the evolving component. It is also the case when it is convenient to modify first the internal structure of a component, as one stage of a larger refactoring, leaving any changes to the interface to later stages.

3.1. Extracting the data link concern

The code related to data links was not modularised, being scattered throughout all classes of the procedure hierarchy (Figure 1) starting with *CompositeProcedure*. This was making it impossible to build a version of WorkSCo devoid of data links, for those clients that do not require such functionality. Therefore data links comprised a good candidate for extraction into its own module.

The data link code comprised 4 types of code sections: fields, methods, code fragments in constructors and code fragments in methods. After creating an empty aspect we dealt with each of these in turn, as prescribed in *Extract Feature Into Aspect* (see 4.1). Moving fields and methods was straightforward and done according to *Move Field From Class To Inter-type Declaration* and *Move Method From Class To Inter-type Declaration* (see 4.2 and 4.3 respectively).

The constructors placed a more complex and interesting problem (Listing 1). Each subclass in the procedure hierarchy adds new arguments for its initialisation so that constructor signatures keep increasing as we go down the inheritance chain. Each constructor makes a super() call passing the arguments defined in the superclasse's constructor and next deals with the initialisation of the data specific to it. Most of these arguments relate to concerns other than the primary one, the data link concern being one example. We wanted all initialisation code related to the data link concern to be placed within the aspect and to keep all other code in the classes. To complicate things, the constructor received arguments related to the concern we wanted to modularise and we were constrained by the decision to maintain its signature for the sake of preserving existing interfaces.

```
public abstract class Procedure
implements Cloneable {
   Procedure(String id,
      String name,
      Precondition precondition,
      IOMessage input, IOMessage output) {
      //initialisation code
   }
   //rest of Procedure code
}

public abstract class CompositeProcedure
extends Procedure {
   protected List _dataLinks = null;
   CompositeProcedure(String id,
         String name,
         Precondition precondition,
         IOMessage input, IOMessage output,
         ArrayList dataLinks) {
      super(id, name, precondition, input, output);
      _dataLinks = dataLinks;
   }
   //rest of CompositeProcedure code
}
```

```
public class SequenceProcedure
extends CompositeProcedure {
   public SequenceProcedure(String id,
         String name,
         Precondition precondition,
         IOMessage input, IOMessage output,
         ArrayList dataLinks,
         List steps) {
      super(id, name, precondition,
            input, output, dataLinks);
      //initialisation code
   }
   //rest of CompositeProcedure code
}
```

Listing 1 – Example of a chain of procedure constructors

We devised *Partition Constructor Signature* (see 4.4) to solve that problem, expecting to apply it the same way as illustrated in the example of section 5. According to that refactoring the base code gets a new constructor with a shorter argument list, devoid of the argument related to the extracted concern, while the aspect introduces a constructor with the original and longer argument list. That introduced constructor makes a this() call to the simplified constructor in the class thus avoiding having code unrelated to the data link concern (Listing 4 shows this technique applied to a simpler example).

When we tried *Partition Constructor Signature* we realised that we were not considering super() calls. They were needed to ensure initialisation code in the superclasses would receive their arguments and run. However, we could not use super() because we were already using this()! As it stood, either the introduced constructors would make the super() calls as in Listing 1, in which case code related to the this() calls would have to be duplicated, or the this() calls would be made, in which case calls to super() could not be made and the related code would have to be duplicated.

For this reason *Partition Constructor Signature*, as presented in section 4.4, applies only to constructors that not pass arguments to super(). In the end we solved the problem by keeping the this() calls in the introduced constructors, just as in Listing 4, and treating the code related to super() as crosscutting code within the aspect, which was extracted to its own advice.

Extracting code fragments from methods also presented some interesting problems, caused by difficulties in capturing the necessary context. The culprits were the large compile() methods through which procedures generate the graph representation of their portions of the workflow. The code responsible for dealing with the data links in each method was placed at the end and was actually larger than the preceding part. At this point we found it useful to apply (JDT's) *Extract Method* ([6], p.110]), not just to ease the way for subsequent refactorings but primarily to isolate the code related to the target concern, thus making it easier to reason with.

Each fragment related to data links used several structures created in the part that preceded it. Those structures were referenced by local variables and neither stored as fields of some object nor passed as arguments. This caused a problem of how to capture them for an advice within the aspect, since AspectJ's pointcut protocol does not cover local variables. We did not have the option of capturing the result of a method because there were several objects, not just one.

It is theoretically possible to capture all the necessary objects, but that would require extremely complex pointcuts, resulting in code hard to understand and error prone. We could obtain through accessors some of the objects after the end of the execution, but at least one complex structure would have to be computed a second time, leading to code duplication (at least performance is not an issue in workflow applications).

Simply turning the local variables into fields so they could be easily captured is a very crude mechanism, but *Replace Method with Method Object* ([6], p.135) offered an organised way to do it. We did apply it, after which using *Extract Advice* (see 4.5) was straightforward. The classes of the method object were created as inner classes within each procedure class. A word of caution, however: contrary to common practice in object programming, we had to leave the fields of the method objects public so that code within the advice could have access to arguments-turned-fields. Otherwise the method objects would need accessor methods, which would be overkill in these circumstances, or the aspect would have to be privileged, something we think should be avoided except as a last resort.

At the start of this experiment we created the aspect in a subpackage of WorkSCo's kernel rather than in the kernel package itself. We regarded the data links functionality as accessory, and to place it in a subpackage makes that explicit. However we started having compiler errors due to visibility violations – several methods and fields referred from code moved to the aspect had restricted access (protected, private or package). We temporarily solved these problems by (reluctantly) classifying the aspect as privileged, until we analysed the various issues and discovered that none of the methods was private – they had either package or protected access modes, meaning they were indeed supposed to be visible, but only within a restricted scope. In the end we kept the aspect in its own subpackage and solved the problems by refactoring the base code, by encapsulating a few fields accessed in the advice code, using *Self Encapsulate Field* ([6], p.171), and relaxing access clauses of some methods called from advice. A class from the kernel was used only by the aspect and it was moved to the data link subpackage.

4. REFACTORINGS FOR FEATURE EXTRACTION

The refactorings presented here were based on the experiment described on the previous section. It should be noted the present collection of refactorings is open ended and does not aim to cover all possible situations, even relative to the subject of feature extraction.

To ensure refactorings are readily applicable we chose a format programmers could recognise, similar to the one used in [6], including the style of cross-referencing the refactorings and mentioning in each case the name and the page number. For this reason we also use an self-contained example that does not mention WorkSCo. Presentations comprise the following elements:

- Name of the refactoring
- Brief mention of a typical situation
- Brief description of the recommended action
- Preconditions (when needed)
- Mechanics
- Code Example

The code examples also use a style similar to the one in [6], with the code fragments subject to the transformations highlighted in bold. These code fragments are taken from the complete example presented in section 5. The refactoring presented first is more high-level than the others. It covers the general feature extraction algorithm and the remaining refactorings refine its various steps.

The transfer of individual members from classes to an aspect should not be taken in isolation. In most cases they are part of a set of transfers that comprise all the implementation elements of a concern being extracted. Such concerns typically include multiple code fragments scattered across multiple modular units (e.g. methods, classes, packages). This is the reason why some of the refactorings presented next are slightly less self-contained than many found in [6].

Some procedure guidelines are generally applicable, independently of the transformation being carried out. These include:

- Ensure the program is adequately unit tested [3][6]. In the absence of automatic tool support there is the danger that transformations will introduce bugs, and the presence of aspects only compounds the problem.
- Facilities from IDEs including search tools are of course recommended: indeed, in large systems they are indispensible.

Keep in mind that most code transformations can potentially break existing pointcuts (even just changing an access clause from public to private). Although we warn of some situations we think it is not feasible to cover them all. Programmers should rely on their knowledge of the code to check potential trouble spots.

4.1. Extract Feature Into Aspect

Typical situation

There is a feature in the base code that is scattered across several units of modularity such as methods and classes. You would like it to evolve separately from the primary code base.

Recommended action

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

Mechanics

- Create an empty aspect in the appropriate package. If the aspect is placed in a separate package, include the host class in the aspect's *import* section.
- Move the concern's various fields to the aspect with *Move Field From Class to Inter-Type Declaration*. 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 Advice*. If some of that code uses some of the constructor's parameters and you want to preserve existing interfaces use *Partition Constructor Signature*.
- Move the concern's various methods to the aspect with *Move Method From Class To Inter-type Declaration*.
- Move any code fragments that do not comprise a full method with *Extract Advice*.
- Change to private the access clauses of all aspect members that became visible only within the aspect.
- Remove the qualifier privileged from the aspect if it no longer accesses non-public members in the primary code.

Example

A complete example of this refactoring is presented in section 5.

4.2. Move Field From Class To Inter-type Declaration

Typical situation

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

Recommended action

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

Mechanics

- If the field is public, consider whether using *Encapsulate Field* ([6], p.206) before this refactoring would be appropriate.
- Move 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.
- Change the field's access clause to public. You can change it to private as soon as all code that deals with the field is placed in the aspect. If for some reason you are forced to leave some code related to the field in class, consider first using *Self Encapsulate Field* ([6], p.146).
- Check for any within() pointcut that should be updated after this refactoring.
- Compile and test.
- For each fragment of code that accesses the field, decide whether the whole method or just a fragment should be moved: (a) use *Move Method From Class to Inter-Type Declaration* for the whole method, (b) use *Extract Advice* for a fragment. You can use *declare warning* to signal occurrences of missed members, as shown next:

```
public aspect WindowView {
     //...
     declare warning:
        get(JTextField TangledStack._text)
        && !within(WindowView):
        "Don't access _text outside aspect.";
     //...
}
```

- Change the field's access clause to private.
- Compile and test.
- Check for any *import* statements that are no longer necessary in the original host class.

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);
   //...
}
```

$\downarrow$

```
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 related to fields is placed in the aspect, change their access clauses back to private and compile and test again:

```
import javax.swing.*;

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

4.3. Move Method From Class To Inter-type Declaration

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.

Preconditions

The most straightforward case 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 met, check for each of the following cases.

a) Check for uses of non-public members 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 Declaration*, next move initialisation code in the constructors with *Partition Constructor Signature*, 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 relax the access. See also if it is a case of moving the aspect to the same package. If you are unable or 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 a second analysis reveals this to be 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 remaining code with *Extract Advice*.

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.

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 unpluggable. 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 Advice*.

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

If the method's access 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 you should be able to change the accesses to private.

Mechanics

- Move the method's definition from the class to the aspect.
- Add the class name and "**.**" before the name of the method.
- If the access is non-public change it (temporarily) to public. As soon as all the code using the method is in the aspect, change it to private.
- Check whether a new *import* statement should be written in the aspect's *import* section.
- Check for any within() pointcut that should be updated after this refactoring.
- Compile and test.

Example

```
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 Declaration* first to 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.

4.4. 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. Note that this refactoring does not apply exactly as presented to constructors making calls with arguments to super() (see 3.1).

Recommended action

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

Mechanics

- Create a new constructor in the class, with a shortened argument list, without the arguments related to the extracted concern.
- Move to the new constructor all the statements not related to the extracted concern.
- Place a call to this() as the first statement in the original constructor, passing only the parameters not related to the extracted concern.
- 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.
- Check whether some poincut targeting the original constructor signature should also cover the new one.
- 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);
    }
    //...
}
```

4.5. Extract Advice

Typical situation

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

Recommended action

Create a pointcut that captures the intended joinpoint and move the fragment of code to the appropriate advice.

Preconditions

Before copying the code fragment a careful analysis of the method'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 the extraction.

A situation that may occur from time to time is the need to capture local variables (either primitives or object references). Such a situation 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, by using *Extract Method* ([6], p.110]) for each part in turn. Such a split may provide the joinpoints you need. In the more complex cases the best option may be to use *Replace Method with Method Object* ([6], p.135). This is the refactoring recommended by Fowler et al to ease the way for *Extract Method*, 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. However keep in mind that the fields of the method object may need to be public.

Mechanics

- Create a named pointcut that captures the intended set of joinpoints. If the intended pointcut is already under construction (from previous uses of *Extract Advice*), extend 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 must be captured. The most usual cases involve the use the target() PCD combined with call(), or the use of this() combined with execution(), set() or get(). Choose a suitable name for the variable. 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();
}
```

- Create the suitable advice for the pointcut, with an empty body (if it is not already under construction).
- Move the code to extract from the source method into the advice's body.
- Add any additional glue code necessary to set up the advice's context.
- Replace references to the self-variable "this" by the 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.

When the advice is meant to replace a large number of scattered fragments you should check which is simpler: to deal with the whole set at a single go or to deal with one fragment at a time. Sometimes the pointcut is complicated to specify when covering only a subset of all 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

```
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();
}
```

5. A COMPLETE EXAMPLE

The small code examples in sections 4.2 to 4.5 illustrate the lower-level refactorings. Here we provide a small complete example that illustrates the higher-level refactoring presented in 4.1 and helps to demonstrate how each of these refactorings fit in the larger picture. Space constraints prevent us from using a more complex example that would cover absolutely all details and issues raised in the previous sections, but the example presented here requires all the refactorings. We also do not present a client program, but care was taken to ensure that the refactorings are transparent to any cient code.

It comprises a stack 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.");
```

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

```
      _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);
   }
}
```

Listing 2 – Stack with two crosscutting concerns

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 Declaration* 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* should be used. This results in two versions of the constructor: the first (argumentless) being placed in the host class and dealing with the remaining concerns (in this case only the primary concern), the other constructor (receiving the JFrame object related to the extracted concern) being placed in the aspect and therefore made unpluggable. 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.

Next *Move Method From Class To Inter-type Declaration* is used to move the method display(), and next use to move the calls to display() in the push() and pop() methods. The declaration "import javax.swing.*;" can be removed from the host class at this point. Finally, we can change to private the access clauses 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 Advice* 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);
   }
}
```

Listing 3 – Stack cleaned of tangled code

```
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();
   }
}
```

Listing 4 – Window view aspect

```
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");
    }
}
```

Listing 5 – Precondition checking aspect

6. DISCUSSION

The experiment described in this report illustrates the convenience of refactoring the (base) code so that it offers the adequate joinpoints. The most important gap in AspectJ's joinpoint protocol seems to be in local variables, and therefore the need to quantify [5] over local variables led to the most intrusive (if structured and disciplined) refactorings. In [15] we call code fashioned this way to be amenable to quantifications *aspect friendly*. Aspect friendly code is subtly different from oblivious code [5] in that programmers are no longer oblivious of aspects but the code itself does not necessarily betray explicit dependencies on aspect constructs.

These issues suggest that new rules of good style must be found for code developed with AOP technology. For instance, we detected a tension between encapsulation and aspects. Finding the desirable style is one task we propose to undertake in future research. The refactorings presented in this report are just the tip of the iceberg and we plan to develop the catalogue as a way to acquire an understanding of the desirable style. We encourage feedback from researchers and practitioners.

It has been noted that standard object-oriented refactorings cannot apply as-is in the presence of aspects [8][9]: changes to base code can easily break the quantification of aspects. Several authors give the name *aspect aware* to object-oriented refactorings that take into account the presence of aspects [8][9]. Iwamoto and Zhao [9] present an analysis of 32 refactorings from [6], concluding that only 3 can be safely used in the presence of aspects. The reason is easy to spot: any refactoring affecting existing joinpoints covered by the pointcut protocol can potentially break aspect code. In [15] we named this problem the *fragile base code problem*. We consider this comprises the strongest case for automatic support for AOP refactoring.

Some papers and articles recently published relate to AOP refactoring. Iwamoto and Zhao [9] propose a number of AOP-specific refactorings but no details are given. Hanenberg et al [8] also present AOP-specific refactorings for feature extraction, besides covering other related subjects. Laddad [13] presents several useful refactorings for extracting various types of concern into aspects and techniques to ensure programmers do not later accidentally change its semantics. None of the above mentioned cover issues with constructors or describe the refactorings in the detailed and systematic presentation format used here.

7. CONCLUSIONS

This report makes the following contributions:

- It describes an experiment in extracting a feature from a Java framework into an AspectJ module.
- It presents in detail and in a familiar format a set of refactorings for extracting crosscutting features into (un)pluggable aspects.
- It presents a complete and self-contained example which places each refactoring in its proper context.

Acknowledgements

We thank António Rito Silva and Ron Bodkin for their feedback and remarks on drafts of this report. Miguel Pessoa Monteiro is partially supported by *PRODEP III (Medida 5 – Acção 5.3 – Eixo 3 – Formação Avançada de Docentes do Ensino Superior)*.

8. REFERENCES

[1] Eclipse Home Page. http://www.eclipse.org/.

[2] Software Engineering Group home page. http://www.esw.inesc.pt/

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

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

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

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

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

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

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

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

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin. Aspect-Oriented Programming. ECOOP'97, Finland, June 1997.

[12] R. Johnson, B. Woolf, Type Object, chapter 4 of "Pattern Languages of Program Design 3" (R. Martin, D. Riehle and F. Buschmann, editors), Software Patterns Series, Addison-Wesley, October 1997.

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

[14] D-A. Manolescu, A Micro Workflow Architecture Supporting Compositional Object-Oriented Software Development, Ph.D. thesis, University of Illinois at Urbana-Champaign, 2001.

[15] M. Monteiro, J. Fernandes, Some Thoughts On Refactoring Objects to Aspects, proceedings of the DSOA'2003 workshop at JISBD 2003, November 2003.

[16] W. F. Opdyke. Refactoring Object-Oriented Frameworks. Ph.D. thesis, University of Illinois, 1992.

[17] M. Robillard, G. Murphy, Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies, ICSE'2002 (pages 406-416), May 2002.