Interfaces, Aspects, and Views

The Discoveries of a Clustering Aspect Miner and Viewer

David Shepherd, Lori Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
{shepherd, pollock}@cis.udel.edu

ABSTRACT

Developers need to be aware of cross-cutting concerns throughout the evolution of a software system in order to make sure they are implemented and updated consistently and correctly. Currently, developers often use interfaces to specify cross-cutting concerns. In many ways, this is good design, as it explicitly links cross-cutting methods across class boundaries, allowing these two decompositions (the class and the interface decompositions) to co-exist. However, because the interfaces do not physically group these methods, the implementation is often inconsistent across classes. In this paper, we describe how our clustering aspect miner and viewer can be used to investigate this kind of cross-cutting concern. We use the evidence that we find to make recommendations of which technology to use in different situations. Our tool uses a distance function to group related code, it provides helpful views of code without noticeable computational demands, and combines aspect mining and viewing research. A case study demonstrates the utility of a clustering viewer for software development and maintenance.

1. INTRODUCTION

Aspect Oriented Programming (AOP) is an exciting new programming paradigm. AspectJ, a leading AOP language, has quickly moved from a research language to a commercial language, without much of the vetting process that a new language usually endures. The quick adoption of AOP is largely due to compelling code examples that researchers have discovered. For instance, persistence, synchronization, and logging involve concerns that are cross-cutting, or implemented across object boundaries. If one attempts to either implement or update that concern, an Aspect Oriented view of the system is undoubtedly helpful.

We call a situation in which a user desires to update or implement a cross-cutting concern a *cross-cutting situation*. In a *cross-cutting situation*, a user would like to have an AOP view of the code, regardless of whether the concern was originally coded in AOP. In order to provide this view, we propose exploiting research in aspect mining to help identify cross-cutting concern code for the user. However, aspect mining research has thus far mainly focused on analyses that return all potential cross-cutting concerns in a system [11, 10, 1, 12, 6, 2]. In a *cross-cutting situation*, an efficient way to find code related to one particular cross-cutting concern is needed.

Researchers have used concept analysis to group code that is related to a particular cross-cutting concern, based on method and class names [14]. The tools group methods and classes that have one or more matching substrings. Results are presented to the user as a list of concept nodes, where each node has a set of children and each child is either a class or a method. Because the actual concept lattice is not reported, many of the relationships between neighboring nodes in the concept lattice are lost from view. Nodes with even small differences appear unrelated in the results view. Views of the method bodies of many methods are not presented simultaneously, but only views of a single method body. Also, because concept analysis is used as the underlying technology, it is uncertain how to move from simpler properties to more complex properties.

Once related code is identified, the user must be presented with the results in a way that facilitates understanding of the cross-cutting concern. Research in aspect refactoring [13, 8, 4] can potentially help move code from many scattered pieces into one aspect. However, the techniques for aspect refactoring do not currently support the automatic refactoring of arbitrary pieces of code into aspects. This type of refactoring, when done in a semantics-preserving manner, can be computationally expensive. A more promising approach is to present the user with a "virtual source file" (similar to an "intentional view" [7]) that represents the cross-cutting concern [3, 9]. This view can avoid the computational demands of refactoring, while retaining much of the benefits of refactoring.

To facilitate a user viewing code that is related to a particular cross-cutting concern, we have developed a natural language processing (NLP) based distance function that is used to cluster related methods. Each cluster contains methods that have a small "distance" between them, and so these groups often represent a cross-cutting concern. We have integrated the mining and the viewing processes into one tool we call the Aspect Miner and Viewer (AMAV).

Section 2 explains our approach, as well as its advantages. In Section 3 we discuss several case studies. In Section 4, we state conclusions and recommendations.

2. APPROACH

2.1 Clustering

We perform agglomerative hierarchical clustering (ALC) in order to group methods [5]. ALC first places every object (in this case, every method in a program) that it will cluster in its own group. Then, it repeats the following:

1. Compare all pairs of groups using a distance function, mark the pair that is the smallest distance apart.
2. If the marked pair's distance is smaller than a threshold value, merge the two groups. Otherwise, stop the algorithm.

Therefore, ALC first places every method in its own group. It then repeats steps 1 and 2 until there are no groups that are closer together than the threshold value. It returns all of the groups whose membership is larger than 1.

Groups are stored as trees. When two groups are merged, a parent node is created, and the two groups are set as the children of the new node. For instance, when groups a and b are merged, a new node c is created, and a and b become children of c. Storing and merging groups in this way creates trees where the leave nodes are more closely related to their neighbors than the nodes near the root.

In order to use the tool, the user triggers clustering of a program, and the results of clustering (a group of hierarchical clusters, or trees) are displayed. Leave nodes are labeled with the name of the method that they represent and the method's type (in parenthesis). Non-leaf nodes are labeled with the common-substring of their children. An example is given in Figure 1. In this example, *doActivity* is a non-leaf node, *UndoRedoActivity* is a leaf node (with its type, UndoRedoActivity, in parenthesis), and *createUndoRedoActivity* is also a leaf node.

Since this analysis is for proof of concept, we used the simplest distance function to group methods. Our distance function, for two methods, m and n, is 1 / commonSubStringLength(m.name, n.name). In order to find the distance between a group and a method, or a group and a different group, the label of the group's root is used in lieu of the method name. An important feature of this distance function is that it is pluggable, which differentiates this work from seemingly similar work related to concept analysis[14]. We can perform clustering on methods with more advanced distance functions in the future; by plugging in a new distance function.

2.2 Viewing

We present the resulting clusters as a lightweight view of code. We believe it is important to see all of the code related to a cluster (or as much as possible) in order to consistently implement the concern related to that cluster. Similar to a "virtual source file", these views show code related to a particular cluster.

AMAV consists of three different panes: the *cross-cutting pane*, the *cluster pane*, and the *editor pane*. The *cross-cutting pane* displays all methods which are related to a cluster's implementations. This pane, although lacking the context of each method (i.e., its class), allows the developer to check the consistency of the concern. The *cluster pane* displays all of the clusters that AMAV found. The editor pane displays the class context (Java file) for a particular method. It allows the user to edit a method's implementation with the cross-cutting and class context available. If a user selects a node in the *cluster pane*, the view that AMAV will generate is a cross-cutting view of all the implementations of the node's children.

Listing 1 shows three panes for a particular application and clustering. The left pane is the *cross-cutting pane*, showing all the implementations of "figureChanged" methods, as if Node "figuredChanged" were selected in the cluster pane. The *editor pane* is showing the context of PertFigure's method "figureChanged", as if PertFigure's "figureChanged" had been selected in the *cross-cutting pane*. This type of view is extremely lightweight to generate, as it involves no analysis (once the clustering is done), only the fetching of text for several method bodies.

2.3 Advantages of Our Approach

Our approach has several advantages that distinguish it from the current state of the art in aspect mining and cross-cutting concern viewing research.

- The distance function is pluggable, leading to easy improvements if the initial experiments are promising.
- AMAV provides helpful views of code without noticeable computational demands.
- AMAV combines mining and viewing research to create an implemented tool. This is a first step towards what aspect mining researchers believe the next generation IDEs should include.

Cross-cutting Pane

CH.ifa.draw.samples.pert.PertFigure

```
public void figureChanged(FigureChangeEvent e) {
    update(e);
}
```

Ch.ifa.draw.figures.TextFigure

```
public void figureChanged(FigureChangeEvent e) {
    updateLocation();
}
```

CH.ifa.draw.contrib.html.HTMLTextAreaFigure

```
public void figureChanged(FigureChangeEvent e) {

    willChange();
    super.basicDisplayBox
       ( e.getFigure().displayBox().getLocation(),
       Geom.corner(e.getFigure().displayBox()) );
    changed();

}
```

CH.ifa.draw.util.GraphLayout

```
synchronized public void
    figureChanged(FigureChangeEvent e) {

    if (nodes!=null) {
       Figure node = e.getFigure();
       if (nodes.containsKey(node)) {
          getGraphNode(node).update();
       }
    }
}
```

CH.ifa.draw.samples.pert.PertFigure (**Editor Pane**)

```
public void update(FigureChangeEvent e) {
    if (e.getFigure() == figureAt(1)) {
       // duration has changed
       updateDurations();
    }
    if (needsLayout()) {
       layout();
       changed();
    }
}
public void figureChanged(FigureChangeEvent e) {
    update(e);
}
public void figureRemoved(FigureChangeEvent e) {
    update(e);
}
public void notifyPostTasks() {
    Iterator iter = fPostTasks.iterator();
    while (iter.hasNext()) {
       ((PertFigure)iter.next()).updateDurations();
    }
}
```

Cluster Pane

```
Node "figureChanged"
    Method figureChanged (PertFigure)
    Method figureChanged (TextFigure)
    Method figureChanged (HTMLTextAreaFigure)
    Method figureChanged (GraphLayout)
```

Listing 1: AMAV panes

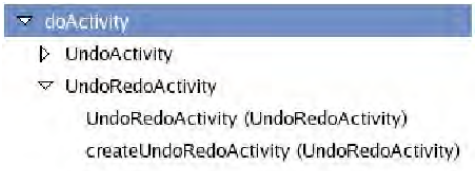

Figure 1: Cluster Pane

- The distance-based measure leverages the concept of distance, grouping the most related code tightly, but still linking (with less weight) slightly less related code. This allows the user to easily extend the boundaries of a concern, leading to less missed code than tools that use boolean relationships.

We implemented AMAV to investigate unknown cross-cutting concerns in a code base. We can easily adapt the tool at a later time to support cross-cutting situations by allowing for a seed (i.e. a starting method name). We will then cluster all method names related to the seed.

3. CASE STUDY

As a case study, we used AMAV to discover cross-cutting concerns in the JHotDraw 5.4b2 code base. Several groups have used JHotDraw to perform aspect mining, and so the source code is familiar to aspect mining researchers and easier to discuss.

In this section, we discuss three representative cases of concerns that we identified and viewed. We discuss one from each of the following categories:

1. Cross-cutting concern represented as an interface with implementing methods that appear consistent
2. Cross-cutting concern represented as an interface with implementing methods that have some code duplication and scattered logic, and appear to be inconsistently implemented
3. Cross-cutting concern represented as methods with similar names across classes, no explicit interface, a large percentage of code duplication, but appear to be consistent

In all three categories, when we refer to implementing methods, we mean the group of methods with the same lexical name, not the entire group of methods required by the interface.

```
Ch.ifa.draw.standard.AbstractFigure
public boolean canConnect() {
    return true;
}
Ch.ifa.draw.samples.pert.PertDependency
public boolean canConnect(Figure start, Figure end)
{
    return ((start instanceof PertFigure) &&
        (end instanceof PertFigure));
}
Ch.ifa.draw.figures.GroupFigure
public boolean canConnect() {
    return false;
}
```

Listing 2: A Little Duplication

3.1 Case 1: Interface with Consistency

This category of cross-cutting concerns is the least common of the three types. It is composed of cross-cutting concerns represented as an interface and its implementing methods, with little code duplication. This cross-cutting concern appears to be implemented in a consistent way across classes. Usually, the kind of interface method that falls in this category is a method whose implementation is extremely class specific. For instance, if there were many different implementors of a Color interface, each implementor's *changeColor* method might be very class specific. The code that is necessary to change a `Car` class's color is likely more complicated and very different than the code to change a `Square` class's color.

Listing 2 presents a lightweight view of the method *canConnect* of the `ConnectionFigure` interface. This view shows the developer that the displayed *canConnect* methods involve no duplication. However, this is an abridged version of the full view (a subset of the *canConnect* methods actually found by AMAV). The cases not shown were very similar to the shown cases, so we did not present them. In JHotdraw, there is a small amount of duplication in the full view, but this remains as one of the few examples of little duplication (when implementing an interface's method) in JHotDraw. Using AMAV, one can quickly recognize that this cross-cutting concern appears to be implemented consistently across classes, since each method follows the pattern of using internal information to decide whether it can connect or not.

3.2 Case 2: Interface with Inconsistency

Listing 1 presents an example of (a subset) of what AMAV clustered around the substring-node *figureChanged*. All children of this node are methods named *figureChanged*. When we use AMAV to look at the implementation of these methods, we notice that one of the methods starts with calls to methods *willChange* and *changed*. This should command the AMAV user's attention, as these types of calls, with related names, which appear at the beginning and end of a method, are usually good candidates for an AOP solution. However, in the other methods, which are implementing the same method for different classes, we do not see these calls.

We thought we might find the calls to these methods by inspecting the methods that are called in *figureChanged*'s method body, so we opened each class in the *editor pane* (by clicking on its method body) in order to explore its class context. To view the methods that we explored see Table 3. There we see that even after following method calls in each classes' context, only three of the four classes call *changed* and only two call *willChange*. Furthermore, the logic to call these methods is tangled in methods other than *figureChanged*. This is an inconsistent implementation of the cross-cutting concern to call *willChange* before a change and *changed* after a change. The scattering of this concern's code has lead to an inconsistent implementation. This inconsistent implementation is quickly identified using AMAV.

3.3 Case 3: No Explicit Interface

The third category involves methods that the developer should have implemented as an interface or an aspect, but did not. Usually, AMAV finds two or more methods that have similar names and a similar implementation. This category, although not as bad as category two, causes concern because logic is repeated in modules that are not explicitly linked. If a developer were to perform a maintenance task on one method, it is likely that he would not know about and therefore not update the related methods.

Listing 4 presents an example of how AMAV identifies this category. The implementation of these methods is almost identical. The first and the third method's implementations are identical, and the second method differs by only one line.

Cross-cutting concerns in categories two and three seem to appear in JHotdraw with about the same frequency, both appear in *at least* several known cases. We suspect there exist many more cases, based on the ease of locating the known cases. Category two is the more disturbing case, because, in the implementation of an interface we find scattered and missing logic and an inconsistent implementation of a cross-cutting concern.

4. CONCLUSION

AMAV uses a distance function to group related code, provides helpful views of code without noticeable computational demands, and combines aspect mining and viewing research. Furthermore, our distance function is pluggable, which will enable us to easily test different distance functions.

AMAV was able to easily reveal several cases in which developers did not implement cross-cutting concerns in a consistent manner. AMAV also exposed cases where concerns were currently implemented consistently, but were in danger of, over time, becoming inconsistent.

CH.ifa.draw.samples.pert.PertFigure

```
public void figureChanged(FigureChangeEvent e) {
    update(e);
}
public void update(FigureChangeEvent e) {

    if (e.getFigure() == figureAt(1)) {
        // duration has changed
        updateDurations();
    }
    if (needsLayout()) {
        layout();
        changed();
    }

}
public void updateDurations() {

    int newEnd = start()+duration();
    if (newEnd != end()) {
        setEnd(newEnd);
        notifyPostTasks();
    }

}
```

CH.ifa.draw.contrib.html.HTMLTextAreaFigure

```
public void figureChanged(FigureChangeEvent e) {

    willChange();
    super.basicDisplayBox
        ( e.getFigure().displayBox().getLocation(),
        Geom.corner(e.getFigure().displayBox()) );
    changed();

}
```

Ch.ifa.draw.figures.TextFigure

```
public void figureChanged(FigureChangeEvent e) {
    updateLocation();
}
protected void updateLocation() {
    if (getLocator() != null) {
        Point p = getLocator().
            locate(getObservedFigure());
        p.x -= size().width/2 + fOriginX;
        p.y -= size().height/2 + fOriginY;
        if (p.x != 0 || p.y != 0) {
            willChange();
            basicMoveBy(p.x, p.y);
            changed();
        }
    }
}
```

CH.ifa.draw.util.GraphLayout

```
synchronized public void
figureChanged(FigureChangeEvent e) {
    if (nodes!=null) {
        Figure node = e.getFigure();
        if (nodes.containsKey(node)) {
            getGraphNode(node).update();
        }
    }
}
void GraphNode.update() {
    Point p = node.center();
    if (Math.abs(p.x - Math.round(x))>1 ||
        Math.abs(p.y - Math.round(y))>1) {
        x = p.x;
        y = p.y;
    }
}
```

Listing 3: View of All Relevant Methods

The views that AMAV provides allow the user to quickly filter through un-interesting clusters and to find clusters that correspond to cross-cutting concerns. The views allow the user to quickly determine whether there is code duplication or cross-cutting concerns in a cluster since AMAV presents the code for all methods within a cluster at one time.

In addition, this research has led to some general rules about when to use interfaces, aspects, or views:

- If, when implementing an interface, there is code duplication in a method's code, move *at least* this duplicated code to an aspect.
- If, when implementing an interface, there is a general policy to enforce (that might not lead to code clones, but similar code), use a view, or even an aspect, when implementing.
- OPINION: Favor views over aspects, as views do not affect the canonical form of the code.

5. REFERENCES

[1] S. Breu and J. Krinke. Aspect mining using event traces. In *Auto. Soft. Eng. Conf.*, 2004.

[2] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe. An evaluation of clone detection techniques for identifying cross-cutting concerns. In *Int. Conf. on Soft. Maintenence*, 2004.

[3] M. C. Chu-Carroll, J. Wright, and A. T. T. Ying. Visual separation of concerns through multidimensional program storage. In *Aspect-oriented Software Development Conf.*, 2003.

[4] R. Ettinger and M. Verbaere. Untangling: a slice extraction refactoring. In *Aspect-oriented software development conf.*, 2004.

[5] S. S. Karanjkar. Development of graph clustering algorithms. Master's thesis, University of Minnesota, 1998.

[6] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In *Working Conf. on Reverse Eng.*, 2004.

[7] K. Mens, B. Poll, and S. González. Using intentional source-code views to aid software

Ch.ifa.draw.standard.ConnectionTool

```
protected Figure findConnectableFigure
(int x, int y, Drawing drawing)
{
    FigureEnumeration fe =
        drawing.figuresReverse();
    while (fe.hasNextFigure()) {
        Figure figure = fe.nextFigure();
        if (!figure.includes(getConnection())
            && figure.canConnect()
            && figure.containsPoint(x, y)) {
            return figure;
        }
    }
    return null;
}
```

Ch.ifa.draw.standard.ChangeConnectionHandle

```
private Figure findConnectableFigure
(int x, int y, Drawing drawing)
{
    FigureEnumeration fe =
        drawing.figuresReverse();
    while (fe.hasNextFigure()) {
        Figure figure = fe.nextFigure();
        if (!figure.includes(getConnection())
            && figure.canConnect()) {
            if (figure.containsPoint(x, y)) {
                return figure;
            }
        }
    }
    return null;
}
```

Ch.ifa.draw.standard.ConnectionHandle

```
protected Figure findConnectableFigure
(int x, int y, Drawing drawing)
{
    FigureEnumeration fe =
        drawing.figuresReverse();
    while (fe.hasNextFigure()) {
        Figure figure = fe.nextFigure();
        if (!figure.includes(getConnection())
            && figure.canConnect()
            && figure.containsPoint(x, y)) {
            return figure;
        }
    }
    return null;
}
```

Listing 4: All Duplication

maintenance. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 169. IEEE Computer Society, 2003.

[8] M. P. Monteiro and J. M. Fernandes. Object-to-aspect refactorings for feature extraction. In *Aspect Oriented Software Development*, 2004.

[9] P. J. Quitslund. Beyond files: programming with multiple source views. In *eclipse '03: Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 6–9. ACM Press, 2003.

[10] D. Shepherd, J. Palm, L. Pollock, and M. Chu-Carroll. Timna: A framework for the combination of aspect mining analyses. In *Technical Report, University of Delaware*, November 2004.

[11] D. Shepherd, L. Pollock, and E. Gibson. Design and evaluation of an automated aspect mining tool. In *Int. Conf. on Soft. Eng. Research and Practice*, 2004.

[12] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Working Conf. on Reverse Eng.*, 2004.

[13] T. Tourwé, A. Kellens, W. Vanderperren, and F. Vannieuwenhuysse. Inductively Generated Pointcuts to Support Refactoring to Aspects. In *AOSD Workshop on Software-engineering Properties of Languages for Aspect Technologies*, 2004.

[14] T. Tourwé and K. Mens. Mining Aspectual Views using Formal Concept Analysis. In *Proceedings of the 4th International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 97 – 106. IEEE Computer Society, 2004.