Using Attribute Slicing to Refactor Large Classes

Douglas Kirk[1], Marc Roper[1], Neil Walkinshaw[2]

[1] The Department of Computer and Information Science, Livingstone Tower, 26 Richmond Street, G1 1XH
[2] The Department of Computer Science, Regent Court, 211 Portobello Street, Sheffield, S1 4DP

Abstract. It can often be the case in object-oriented programming that classes bloat, particularly if the represent an ill-formed abstraction. A poorly formed class tends to be formed from disjoint sets of methods and attributes. This can result in a loss of cohesion within the class. Slicing attributes can be used to identify and make explicit the relationships between attributes and the methods that refer to them. This can be a useful tool for identifying code smells and ultimately refactoring. Attribute slicing can also be used to examine the relationships between attributes, as is the case in decomposition slicing. This paper introduces attribute slicing in the context of refactoring bloated classes.

1 Introduction

This paper introduced the concept of attribute slicing - a form of decomposition slice [8] based on the attributes (also known as fields or instance variables) of a class, and its application to design-flaw detection and refactoring (transformation) [5]. The motivation for this concept originated from desire to be able to identify and split a 'large' class (i.e. one that represented an ill-formed abstraction) on the basis of the usage made by the methods of the attributes of the class. This is based on the observation that if most attributes are not used by most methods (i.e. there are subsets of methods which use distinct subsets of attributes) then the class is an unhealthy composition of abstractions and should be refactored into a number of smaller classes.

1.1 Motivating Problem

The definition of an object in the form of a class captures the state and behaviour of the object. This state and behaviour should all be strongly related in order for the class to be cohesive. Striving for cohesion and ensuring that the class implements just a single abstraction is an important and well-established design guideline that ensures that the system is well-balanced and the class is easy to understand and usable in multiple contexts in the system. However, this ideal is not always easy to achieve and it is even harder to maintain as the system grows and the design evolves. This leads to an imperfectly designed system which may exhibit several design flaws or "bad smells" [5].

Dagstuhl Seminar Proceedings 05451Dagstuhl Seminar Proceedings 05451
Beyond Program Slicing
http://drops.dagstuhl.de/opus/volltexte/2006/490

One of the most prolific bad smells is referred to as a "god class". Besides having many methods and lines of code, a god class is a code smell because it captures more than one abstraction; its functionality is implemented by methods that do not strictly belong together causing poor cohesion. This may have happened for a number of reasons - a poor early design decision, a consequence of several small evolutions - the precise reasons are not relevant. This paper identifies a slicing approach that is useful for identifying this problem as well as the statements in the source code that need to be refactored.

We have observed that the way in which the methods reference the class attributes serves as a strong indicator of whether or not a class is a god class. In a cohesive class the state and functionality are closely related and most of the methods manipulate (access or mutate) most[3] of the instance variables. In a class which exhibits a lack of cohesion this uniformity of attribute usage begins to fragment and distinct subsets of method/attribute usage begin to appear. This is a classic symptom of a class which implements more than one abstraction - the additional methods require additional state and have little interaction with the rest of the class. The role of attributes in the identification of poor class abstractions has been noted in several real systems.

2 Attribute Slicing

The approach taken in this paper is to use the notion of Attribute Slicing as a mechanism for both identifying and addressing the "God class" code smell. An attribute slice identifies the code that is related to (uses or modifies) a particular attribute, or set of attributes in a class (in this case it can be perceived as a decomposition slice, this is elaborated in section 3). As a starting point we need to modify the definition of Weiser's conventional slice [12]. A conventional slice criterion consists of a set of variables and a single point on the control flow graph. With an attribute slice, the attribute is not tied to a specific point in the control flow graph. If we consider the attribute to be the target then:

- A backward slice identifies those statements that affect the value of the attribute.
- A forward slice identifies those statements that are affected by the value of the attribute.

Applying these ideas to the simple case below (and taking some syntactical liberties) these slices are illustrated in figure 1. Note that the backward slice essentially extracts the mutators and the forward slice the accessors. The methods `incx()` appears in both cases since it both accesses and mutates the attribute. By performing both backwards and forwards slicing the entire class is returned in this case.

[3] It would be nice to be able to be definitive in these observations and say "all" rather than "most" but software design, in common with any other human task, involves judgement and compromise so it is rarely possible to make such absolute statements.

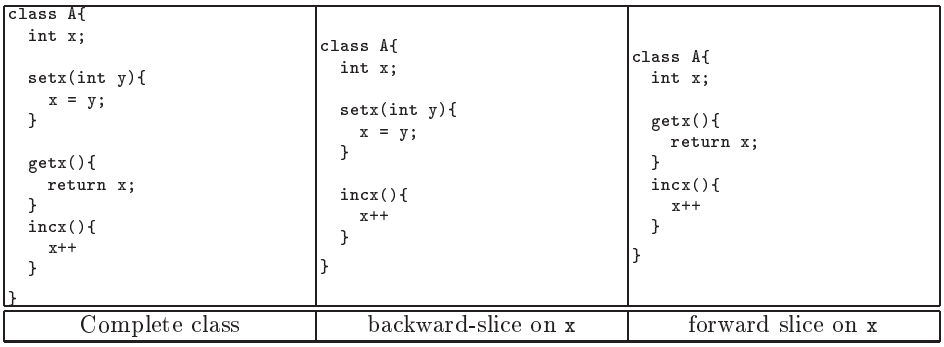


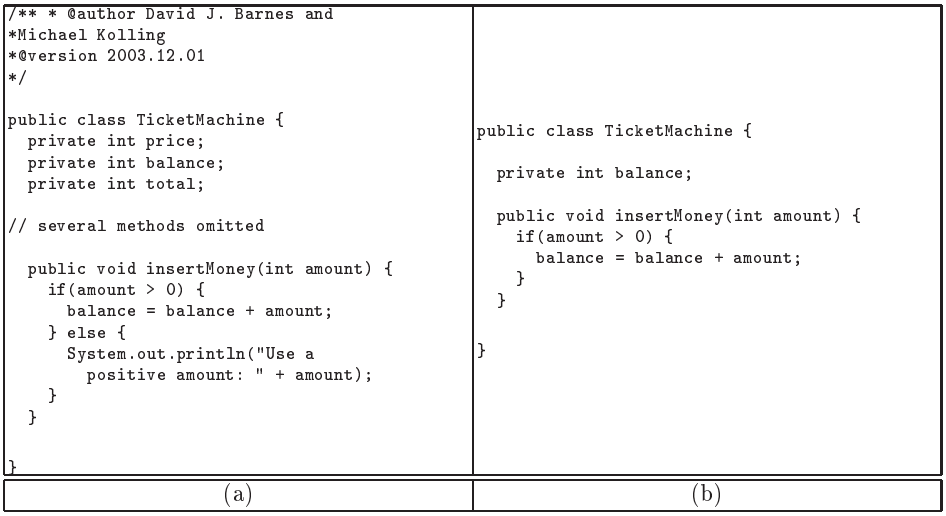
Fig. 1. Illustration of backward and forward attribute slices

So far, nothing has been said about the details of the slice construction or the level of abstraction at which this is applied. The example in figure 1 is very simple and entire methods are returned as part of the slice. Attribute slicing has the potential to be applied at the method level and the intra-method level: A method level attribute slice returns the entire body of a method if it contributes to the slice in any way (i.e any part of the method updates or manipulates the attribute). An intra-method level attribute slice takes into account the detailed control structure of a method and, in the same vein as traditional slicing, returns only the code that updates or manipulates the attribute.

Figure 2 serves to illustrate the difference between method and intra-method level attribute slices. Figure (a) contains the source code for a simple ticket machine class (taken from Barnes and Koelling [1], the class contains several other methods but these are omitted to aid this discussion). If we slice (either forwards, backwards or both) on the `balance` attribute, the entire `insertMoney` method will be returned (along with the `balance` attribute). Sometimes however we need more precise information, and so an intra-method slice would return only the code that concerns the balance attribute, as shown in figure (b). In this case the backwards and forwards slices are identical, but this is not generally the case.

3 Viewing Attribute Slices as Decomposition Slices

Having carried out slices on the various attributes of a class we can then investigate the way in which an attribute contributes (or not) to the general cohesion of the class by applying decomposition slicing [8]. Decomposition slicing is an attractive technique for this problem, because it makes explicit the relationships between the attributes (i.e. whether or not and to what extent the computation of attribute x is related to the computation of y). To recap, a decomposition slice DS(x) is not taken with respect to a single point in the program, but only with respect to a variable x. It contains those lines that can affect the value of x at



```
/** * @author David J. Barnes and
*Michael Kolling
*@version 2003.12.01
*/

public class TicketMachine {
  private int price;
  private int balance;
  private int total;

// several methods omitted

  public void insertMoney(int amount) {
    if(amount > 0) {
      balance = balance + amount;
    } else {
      System.out.println("Use a
        positive amount: " + amount);
    }
  }

}
```

(a)

```
public class TicketMachine {

  private int balance;

  public void insertMoney(int amount) {
    if(amount > 0) {
      balance = balance + amount;
    }
  }

}
```

(b)

Fig. 2. Illustrating method level and intra-method level attribute slices

"output points" in the program. Once decomposition slices are calculated with respect to all of the variables (attributes in our case) the decomposition slice contents are compared (using set relationships) and each decomposition slice is divided into the following three parts:

- The **independent** part: Statements that belong to the decomposition slice and no other decomposition slices.
- The **dependent** part: Statements that belong to the decomposition slice and also belong to other decomposition slices.
- The **complement**: Statements that don't belong to the decomposition slice.

Gallagher and Lyle observe that by manipulating a statement that belongs to the dependent part, it can't affect the behaviour of any statements belonging to the complement. For us this is interesting because the extent to which an attribute is involved with the rest of the class is determined by the contents of its dependent part. If the dependent part is empty (an extreme case), we have a strong case for removing the attribute from the class. If it isn't, we can find out which other attributes it is involved with by comparing the contents of its dependent part with the dependent parts of the rest of the decomposition slices. It may also be feasible to use this as a basis for quantifying how much a particular attribute contributes to the cohesion of the class in general.

Figure 3 shows how viewing attribute slices as decomposition slices can be useful for establishing (a) the relationships between attributes and (b) the appropriate refactoring (if any). The venn diagram shows which statements belong

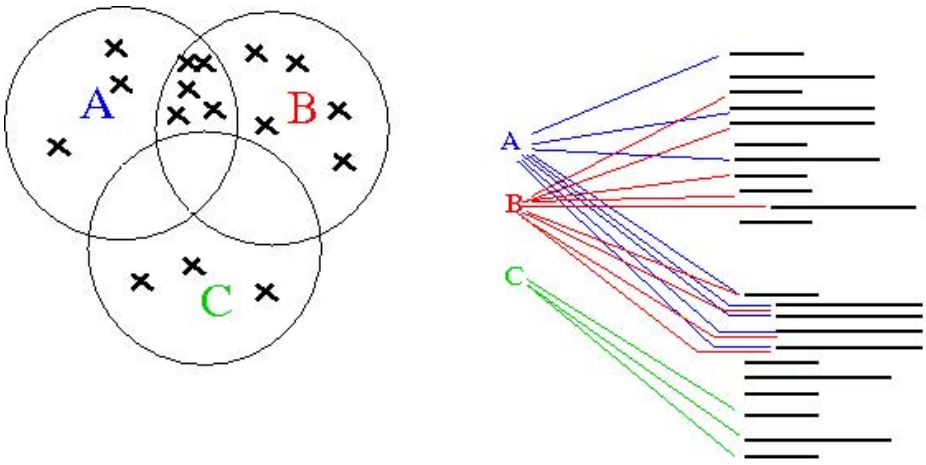


Fig. 3. Investigating relations between attributes A, B and C

to which attribute slices[4], making the independent part, dependent part and complement explicit. In figure 3 it becomes apparent that the dependent part of the attribute slice on variable c is empty. This indicates that variable c has no relationship to variables a and b. In terms of establishing suitable refactorings this is useful in two respects. Firstly, it indicates that variable c is not integral to the functionality of the class as a whole. Secondly, it suggests that the removal of those statements that manipulate and are manipulated by attribute c (e.g. via the 'move field' refactoring' [5]) will have no effect on the functionality that is related to attributes a and b.

4 Application to Refactoring

At the start we mentioned that this was motivated by the desire to investigate splitting large classes that had been formed from more than one abstraction. To illustrate this application consider the following pathological case - the StudentHeater (an unfortunate alliance between a simple student class and a heater thermostat) adapted again from Barnes and Kollings' book [1]. The plain source code is shown in figure 4 (a) and the slices are shown in (b). The slices are colour-coded (so those pertaining to name are in `yellow`, those for id are in `blue` etc.), and both backward and forward slicing has been applied at the method level. Methods that belong to multiple slices are highlighted in a different color and annotated.

[4] These attribute slices are a union of forward and backward slices on attributes, i.e. they contain both those statements that can affect and can be affected by the values of the attributes.

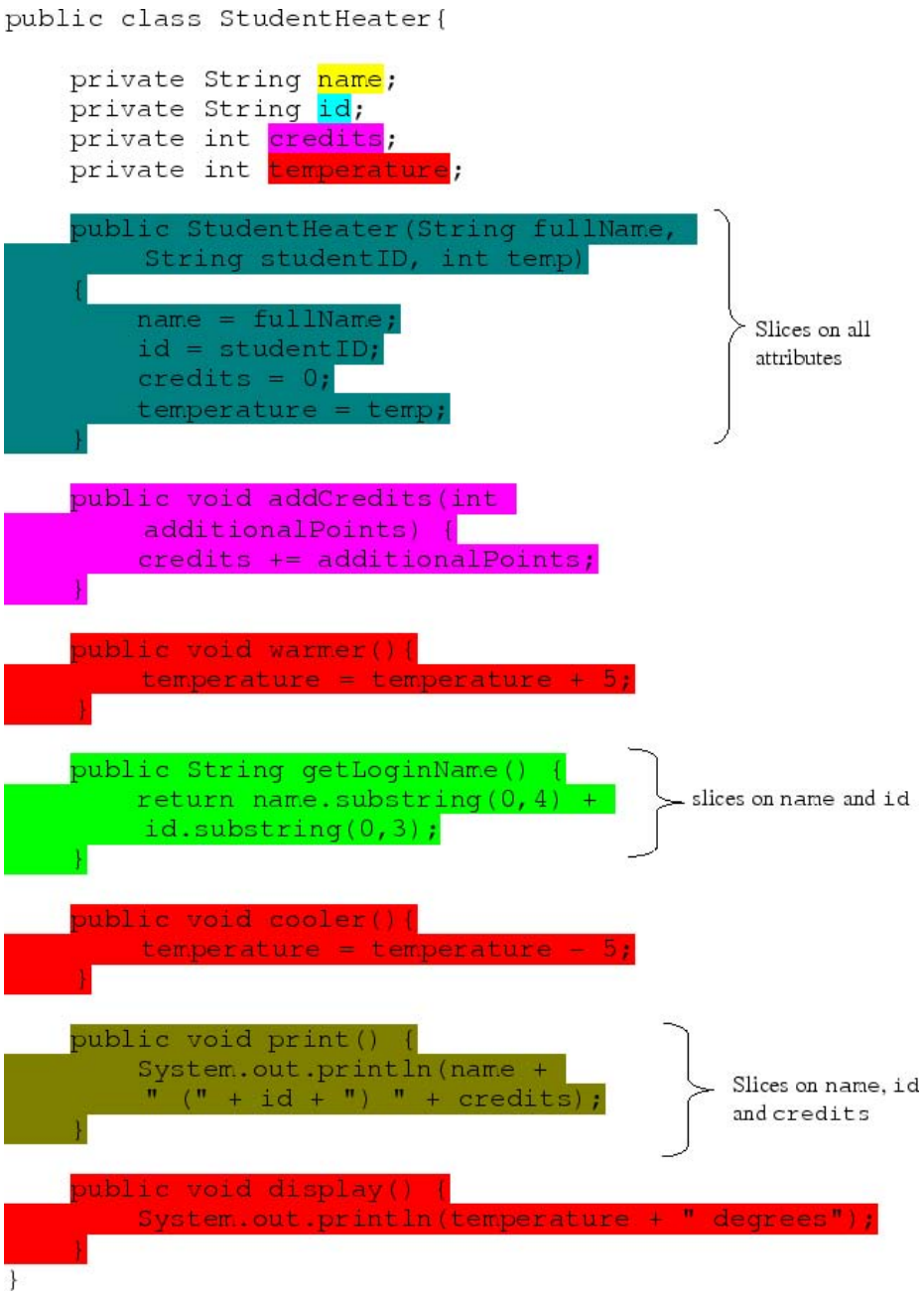


Fig. 4. StudentHeater example

In trying to detect problems in classes, with a view to splitting a class up according to its attributes, we are interested in identifying disjoint or nearly disjoint subsets of slices depending on what is being sought (sometimes, for example, the slice will be on a set of clearly related attributes). It is clear from example in figure 4 that the `temperature` attribute (unsurprisingly) shares little in common with the other attributes except in the constructor. If, upon further analysis, we are convinced that `temperature` should be factored out of this class then we can immediately extract the methods in red, and then apply intra-method slicing to extract the code relevant to the temperate attribute for the shared methods (the constructor in this case, but potentially other methods too), leaving the complement (as in decomposition slicing) in place to form a student class. Although the above case is deliberately extreme we have witnessed several much larger and complex cases which exhibit similar properties.

5 Further Observations and Complicating Issues

This section illustrates some of the potential problems we have encountered when trying to refactor large classes. These are not insurmountable problems but are important to consider, especially when performing any automatic analysis and transformation.

5.1 Dependencies between attributes

This seems to occur widely in the classes we have looked at. In particular we see this as a problem not just of dividing the code into separate classes but also of preserving the ordering of actions between the new classes. In particular it seems to mandate that the original method remain in some form so that the classes can share a dialogue which matches the original sequence - this is highly undesirable for refactoring as it keeps the original class in existence and it also maintains its size.

```
public void addToSelection(Figure figure) {
  if (!fSelection.contains(figure)) {
    fSelection.addElement(figure);
    fSelectionHandles = null;
    figure.invalidate();
    selectionChanged();
  }
}
```

Fig. 5. Dependence between `fSelection` and `fSelectionHandles` in JHotDraw

Figure 5 illustrates how dependence can occur between attributes in an example taken from JHotDraw. The sequence in which these dependences occur is important if the resulting refactoring is to preserve the semantics of the class. This method contains direct statements acting on both the `fSelection`

and `fSelectionHandles` attributes. At first glance this doesn't appear to be a problem as it doesn't particularly matter whether one happens before the other but what is not obvious is that the final method call - to `selectionChanged` - is indirectly accessing the `fEditor` attribute (telling it to redraw its menus with respect to the changes in the selection). This is dependant upon the change to the `fSelection` attribute and must come after it for the behaviour of the system to be preserved.

5.2 Methods that exhibit no direct attribute usage

In our experience these mainly fall into two camps: Methods that access only local variables to compute something, and methods that indirectly use (private) methods to access attributes.

Methods that access only local variables The problem that arises in this case is where to put them if the class is to be refactored. Do they belong with other methods associated with an attribute or attributes? If so how can this be established?

The problem is illustrated in figure 6. This might be a somewhat pithy example as handles are perhaps unusual but it does show that methods can exist in a class without affecting its state directly. It could perhaps it could be argued that the handles vector is a virtual attribute of the class as it is dynamically created there during execution.

```
public Vector handles() {
  Vector handles = new Vector();
  handles.addElement(new NullHandle(this, RelativeLocator.northWest()));
  handles.addElement(new NullHandle(this, RelativeLocator.northEast()));
  handles.addElement(new NullHandle(this, RelativeLocator.southWest()));
  handles.addElement(new NullHandle(this, RelativeLocator.southEast()));
  return handles;
}
```

Fig. 6. Method from `StandardDrawingView` in JHotDraw that only accesses local variables

Methods that use private methods to access attributes Here the problem again is how to best transform the existing code if the class needs to be refactored. Does the method belong with the attribute class (i.e. it exclusively accesses one attribute indirectly), or can it be split up in the face of multiple shared attributes, or should it in fact exist in some other as yet unidentified class accessing both attributes through separate public interfaces? In general it appears difficult to resolve this problem from the static relationships of the code alone (although it can be detected easily enough). Related to this it is worth noting that methods which are declared as static in Java (i.e. class methods) may need some form of

special treatement. The example in figure 5 already contains an illustration of an indirect usage (where `selectionChanged` hides use of the `fEditor` variable).

5.3 Inheritance

Inheritance effectively distributes attributes across a hierarchy of classes. Some attributes are protected (thereby breaching encapsulation) and some are private. This creates a problem not just because splitting into attribute classes might effect multiple places in the original code but also because dependencies can exist between public/protected and private attributes which may make them harder to split apart.

One case observed is in `StandardDrawing`, where the class implements the `Drawing` interface but inherits from `CompositeFigure`. This example is interesting as the `CompositeFigure` provides about half of the implementation of the `Drawing` interface even though it is not supposed to be implementing any of it (it is higher up the hierarchy but obviously has been created in anticipation of becoming a part of the `Drawing`). It seems that composition rather than inheritance is called for to help keep the interfaces focused on the domain abstractions they are supposed to represent.

5.4 Dependent Clients

It is unlikely that all the code relegated to an attribute will exist nicely within the class containing that attribute. There are a number of bad smells which allude to the idea of code out with the class feeling envious and wanting access. This raises the question of how to check for and find this misplaced functionality in the classes that talk to the target in question, and this is achievable by extending the slice outside the bounds to the class to pick up these dependent clients. This is bad for the usual reasons that it introduces the possibility of polymorphism and basically changes the re factoring from a local search problem into a global one.

An example of functionality being distrusted outside of a class occurs between the drawing and the view in JHotDraw. In one case the view should be responsible for displaying the contents of a drawing yet delegates the rendering of its contents to the drawing. This is clearly a breach of MVC as the model is now dictating what the view will look like. Instead, the draw method of drawing should be moved over to the view so that it can control how the figures are rendered (this also affects figures as the drawing delegates to them to render their contents).

Similarly there is a method in the view which makes more sense in the drawing. The `checkDamage` method gets the listeners of a drawing and searches through them looking for drawing views - when any are found they are sent a message telling them to redraw. This should not be the responsibility of the drawing view instead the drawing should be monitoring its state changes and whenever it feels that a redraw is required it should ask all its views to redraw. Its interesting to note that both cases where a method has been detected out of place are design

patterns; perhaps it is knowledge of the expected structure that is helping to detect the misplaced functionality.

6 Related Work

This work is closely related to work on metrics and slicing. A substantial amount of work has been carried out (primarily by Bieman *et al.*) into the use of slices to compute cohesion. Their work is elaborated in section 6.1. Section 6.2 looks at the relevant slicing-related research, concentrating on decomposition slicing and its use in software maintenance.

6.1 Measuring Cohesion

Cohesion is notoriously difficult to measure and has been the subject of a substantial amount of research. Most of this work has been carried out in the procedural domain. Bieman and Ott [2] investigated the use of slices to compute functional cohesion. They produce a slice-based technique that can be used to measure the cohesion of individual procedures (based on the overlap of the slices for variables in the procedure).

In later work [10] they reconsider the notion of cohesion when applied to object-oriented systems. Citing Fenton [4], they establish an interesting dichotomy between the (traditional) notion of functional cohesion, which cannot be applied directly to object-oriented classes, and a new notion of cohesion called *data cohesion.* They extend the original cohesion computation approach illustrated in [2] and extend it by computing slices with respect to the attributes. The end result of their procedure produces two measures: Strong data cohesion and weak data cohesion. The former measure counts the number of statements that belong to *all* of the slices.

Their motivation for computing slices of attributes is similar to ours; they want to identify (lack of) cohesion in classes. There are however important differences between their work and ours. They simply aim to establish the extent of class cohesion, returning an absolute value. Our approach aims to use the slices not only as a means to obtaining the extent of class cohesion, but also aims to make the slices themselves a resource for determining which elements of the source code can be (safely) altered during the refactoring process and how they are related to each other (via decomposition slicing, see below).

6.2 Decomposition Slicing

Decomposition slicing was proposed by Gallagher and Lyle [8]. It provides a framework to compare the contributions made by a set of variables to the functionality of the program as a whole (see section 3). It has been implemented by Gallagher *et al.* in their *Surgeon's Assistant* [6] and has been evaluated extensively on procedural source code. This is (to the best of the authors' knowledge)

the first paper to propose the use of (a specialised form of) decomposition slicing to analyse the composition of classes by comparing slices on class attributes.

An inherent problem that arises in using slices as feedback for users is that, despite the usefulness of establishing *which* statements belong to a slice, it is very difficult to convey why a given statement belongs to the slice. Our work aims not only to establish which statements belong to attribute slices, but also to convey to the programmer *how* a given attribute contribute to the functionality of the class as a whole. Gallagher [7] suggests a visual solution to this problem, where the relationship between decomposition slices can be visualised as a graph that shows a partial ordering between them (and hence how they are related to each other). Tonella [11] has recently elaborated on this work, using a formal concept lattice (which is a product of formal concept analysis [9]) to produce a lattice of decomposition slices that includes nodes that suggest points of interference between variables that are not apparent on Gallagher's graphs.

6.3 Slicing for Refactoring

Ettinger and Verbaere [3] have developed an Eclipse-based tool that can be used to refactor Java programs. Their work is an important demonstration of the potential of using slicing to refactor code. Their slicing primarily aims to automate the transformations at a method-level (i.e. extracting a new method from an existing method) and does not concentrate on identifying code smells at a higher, structural level, as is the case with attribute slicing.

7 Conclusions and Future Work

This paper has described the problem of identifying and splitting large classes through the application of attribute slicing and the investigation of the relationships between the slices. The notion of attribute slicing has been illustrated and the principle of the technique explained. The work is very much in its early stages and to progress it further the following work is planned. Firstly, the notion of attribute slicing and its variations needs to be formally defined. Secondly, the technique requires implementation. At the more abstract method level this appears to be fairly straightforward and much of the computation can be carried out using existing tools such as Eclipse. The intra-method slices on the other hand may be more challenging and will need a program dependence-based representation of the class. Finally, the technique needs to be applied to larger examples both to validate its accuracy and identify any potentially interesting cases that may have been overlooked.

References

1. D. Barnes and M. Koelling. *Objects First With Java - A Practical Introduction Using BlueJ.* Prentice Hall / Pearson Education, 2004.

2. J. Bieman and L. Ott. Measuring Functional Cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–658, August 1994.
3. R. Ettinger and M. Verbaere. Untangling: A Slice Extraction Refactoring. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD'04)*, 2004.
4. N. Fenton. *Software Metrics - A Rigorous Approach*. Chapman and Hall, 1991.
5. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
6. K. Gallagher. Evaluating the Surgeon's Assistant: Results of a Pilot Study. In *Proceedings of the International Conference on Software Maintenance (ICSM'92)*, 1992.
7. K. Gallagher. Visual Impact Analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM'96)*, 1996.
8. K. Gallagher and J. Lyle. Using Program Slicing in Software Maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.
9. B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.
10. L. Ott, J. Bieman, B-K. Kang, and B. Mehra. Developing Measures of Cohesion for Object-Oriented Software. In *Proceedings of the Annual Oregon Workshop on Software Metrics (AOWSM'95)*, 1995.
11. P. Tonella. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. *IEEE Transactions on Software Engineering*, 29(6):459–509, 2003.
12. M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.