Aspect Mining through the Formal Concept Analysis of Execution Traces

Paolo Tonella and Mariano Ceccato
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{tonella, ceccato}@itc.it

Abstract

The presence of crosscutting concerns, i.e., functionalities that are not assigned to a single modular unit in the implementation, is one of the major problems in software understanding and evolution. In fact, they are hard to locate (scattering) and may give rise to multiple ripple effects (tangling). Aspect Oriented Programming offers mechanisms to factor them out into a modular unit, called an aspect.

In this paper, aspect identification in existing code is supported by means of dynamic code analysis. Execution traces are generated for the use cases that exercise the main functionalities of the given application. The relationship between execution traces and executed computational units (class methods) is subjected to concept analysis. In the resulting lattice, potential aspects are detected by determining the use-case specific concepts and examining their specific computational units. When these come from multiple modules (classes) which contribute to multiple use-cases, a candidate aspect is recognized.

1 Introduction

Object-Oriented systems are developed by mapping the real-world entities of the application domain to a hierarchy of classes, around which the software is developed. We will call such class organization the *principal decomposition* of the system. Not all the requirements of the application under development correspond necessarily to a single modular unit (class) in the principal decomposition. The implementation of some requirements may be scattered across the classes and possibly mixed with the functionalities that implement the other responsibilities of the classes. Such a mismatch between requirements and implementation is a potential source of serious problems during software maintenance. In fact, it may be difficult to trace the requirements to the modular units implementing them and correspondingly it may be hard to evolve the code when the requirements change.

Aspect Oriented Programming aims at eliminating the mismatch between transversal requirements (crosscutting concerns) and the code by providing a modular unit, called an *aspect*, where they can be located. The implementation of a crosscutting concern is part of a unique module, an aspect, instead of being scattered across and tangled with several classes. The requirements that crosscut the principal viewpoints are discussed in detail in [**?**].

Modularization of crosscutting concerns into aspects is potentially beneficial for existing software systems. However, the hard job is deciding what functionalities should be regarded as aspects. Such a problem is called the *aspect mining* problem. In this paper, we propose to use dynamic analysis in order to exercise the computational units involved in the main application functionalities. Then, the relationship between the execution traces associated with such functionalities and the computational units (class methods) invoked during each execution is examined, by exploring the concept lattice produced by formal concept analysis. In such a structure, it is possible to isolate the nodes (concepts) that are specific of a single functionality and to determine which classes are specifically involved (feature location). When a class contributes to multiple functionalities together with other classes, the presence of a potential crosscutting concern is spotted and the opportunity of migrating some code portions to an aspect is detected.

Application of the proposed aspect mining method to a small but meaningful Java example gave very positive and encouraging results. A crosscutting functionality was factored out into an aspect, which could be untangled from the original code thanks to the possibility of redirecting the original execution flow to the aspect code. A one-to-one mapping was achieved in this way between the functionality migrated to the aspect and the module containing it, with an expected positive effect on the code understandability and maintainability.

The paper is organized as follows: after a brief summary of feature location based on concept analysis, given in Section 2, our proposal of a novel aspect mining method is described (Section 3). Its application to a case study is discussed in Section 4, followed by the related works (Section 5) and by our concluding remarks (Section 6).

2 Concept analysis for feature location

In this section, the method for feature location based on concept analysis is summarized, in order to make the paper self-contained. The interested reader can find more details in [3].

2.1 Concept analysis

Concept analysis [5] is a branch of lattice theory that provides a way to identify maximal groupings of objects[1] that have common attributes. Given a *context* (O, A, R), comprising a binary relationship R between objects (from the set O) and attributes (from A), a *concept* c is defined as a pair of sets (X, Y) such that:

$$X = \{o \in O | \forall a \in Y : (o, a) \in R\} \tag{1}$$

$$Y = \{a \in A | \forall o \in X : (o, a) \in R\} \tag{2}$$

where X is said to be the *extent* (*Ext*$[c]$) of the concept c and Y is said to be its *intent* (*Int*$[c]$).

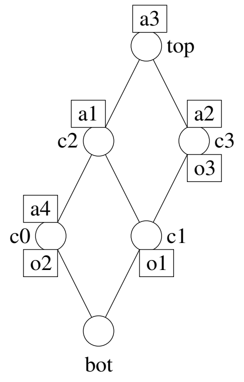
Figure 1 shows an example of context (top), together with the concepts that can be computed for it (bottom).

The containment relationship between concept extents (or, equivalently, intents) defines a partial order over the set of all concepts, which can be shown to be a lattice [5] (see Figure 1, middle). Intuitively, the sub-concept relationship represented in the concept lattice can be interpreted as a specialization of more general notions. In fact, super-concepts have larger extents (and smaller intents) than sub-concepts. Thus, while super-concepts group together objects based upon a small set of shared attributes, sub-concepts aggregate less objects constrained by a larger set of attributes in common. They are thus more specific instances of general notions (the super-concepts).

Complete information about each node n in the concept lattice L is given by the pair (*Ext*$[n]$, *Int*$[n]$). However, it is possible to represent the same information in a more compact and readable form by marking a node n with an object $o \in$ *Ext*$[n]$ or an attribute $a \in$ *Int*$[n]$ only if it is associated with the most special (respectively, general) concept c having o (resp., a) in the extent (resp., intent). The (unique) node of L marked with a given object o is thus:

[1]Not to be confused with objects in Object-Oriented programming.

	a_1	a_2	a_3	a_4
o_1	×	×	×	
o_2	×		×	×
o_3		×	×	



$$\begin{aligned}
bot &= (\{\}, \{a_1, a_2, a_3, a_4\}) \\
c_0 &= (\{o_2\}, \{a_1, a_3, a_4\}) \\
c_1 &= (\{o_1\}, \{a_1, a_2, a_3\}) \\
c_2 &= (\{o_1, o_2\}, \{a_1, a_3\}) \\
c_3 &= (\{o_1, o_3\}, \{a_2, a_3\}) \\
top &= (\{o_1, o_2, o_3\}, \{a_3\})
\end{aligned}$$

Figure 1. *Example of concept lattice.*

$$\gamma(o) = \inf\{n \in L | o \in Ext[n]\} \tag{3}$$

where *inf* gives the infimum (largest lower bound) of a set of concepts. Similarly, the unique lattice node marked with a given attribute a is:

$$\mu(a) = \sup\{n \in L | a \in Int[n]\} \tag{4}$$

where *sup* gives the supremum (least upper bound) of a set of concepts. The objects in the extent of a lattice node n are then obtained as the set of objects at or below n, while the attributes in its intent are those marking n or any node above n (see Figure 1, middle).

The labeling introduced by the functions μ and γ give the concept most specific of a given attribute/object. Thus, given a concept c, the objects and attributes that label it indicate the most specific properties (objects and attributes) that characterize it.

2.2 Feature location

The goal of feature location [3] is to identify the computational units (e.g., procedures, class methods) that specifically implement a feature (e.g., requirement) of interest. Execution traces obtained by running the program under given scenarios provide the input data (dynamic analysis).

Concept analysis is applied to a context where attributes are computational units, objects are scenarios and the relationship R contains the pair (o, a) if the computational unit a is executed when scenario o is performed. Information about which computational unit is executed for each scenario is obtained by gathering execution traces of the program.

A concept in the resulting concept lattice groups all computational units executed by all scenarios in the extent. Moreover, a computational unit labels a given concept if it is the most specific computational unit for the scenarios in the concept extent. Assuming that each scenario is associated with exactly one feature (the general case of a many-to-many relationship between scenarios and features is dealt with in [3]), the concept specific of a given feature is the one (if any exists at all) which has only the associated scenario in its extent (e.g., concept c_0 for the scenario o_2 in Figure 1). Correspondingly, the computational units specific for a given feature are those which label such a concept (a_4 for c_0).

When a feature-specific concept does exist, the attributes that label it are the most specific computational units involved in the execution of the scenario in its extent. In other words, they represent the code portions that are most specifically devoted to implementing the functionality exercised by the scenario in the concept extent.

3 Aspect mining

Aspect Oriented Programming (AOP) [9] is a new programming paradigm, with constructs explicitly devoted to handling crosscutting concerns. In an Object-Oriented system, it often happens that functionalities, such as persistence, exception handling, error management, logging, are scattered across the classes and are highly tangled with the surrounding code portions. Moreover, the available modularization/encapsulation mechanisms fail to factor them out. Aspects have been conceived to address such situations. AOP introduces the notion of *aspect*, as the modularization unit for the crosscutting concerns. Common code that affects distant portions of a system can be located in a single module, an aspect. The aspect compiler takes care of weaving it with the affected code to produce the executable.

Aspects interact with the given code by means of two main mechanisms: pointcuts and introductions. Pointcuts intercept the normal execution flow at specified join points. Aspect code (called *advice*) can be executed either before, after or instead of (around) the intercepted join point. Introductions modify properties of the classes they affect by adding methods or fields and by making them implement interfaces or specialize super-classes previously unrelated with them.

The following aspect introduces a field (*isLocked*) and a method (*setLock*) into class *A*. Moreover, it intercepts any execution of methods whose name start with *action* on objects (named *a*) of class *A*, each time *isLocked* is true. The execution is replaced with the print of an error message, by means of an around advice:

```
aspect Lock {
   boolean A.isLocked = false;
   public void A.setLock(boolean lock) {
      isLocked = lock;
   }

   pointcut lockedExecutions(A a): execution(void A.action*(..))
      && this(a) && if (a.isLocked);
   void around(A a): lockedExecutions(a) {
      a.printError("Selected action is currently locked");
   }
}
```

The first step in the migration of existing applications to AOP consists of identifying the crosscutting concerns that are amenable for an aspect-oriented implementation. Such a process, called *aspect mining*, can be driven by the use cases of the application. In fact, each use case specifies a functionality of the system. When such a functionality is implemented by code fragments spread across several modularization units, it is possible to turn it into an aspect. In the restructured code, each distinct functionality will be located in exactly one modularization unit. However, benefits in program understanding and evolution can be actually achieved only if the new modularization units that factor out the crosscutting concerns are relatively independent (decoupled) from the other units. In other words, restructuring should aim both at separating the crosscutting concerns (increased intra-module cohesion) *and* at untangling them from the original code (reduced inter-module coupling).

We propose to use feature location for aspect mining according to the following procedure. Execution traces are obtained by running an instrumented version of the program under analysis for a set of scenarios (use cases). The relationship between execution traces and executed computational units is subjected to concept analysis. The execution traces associated with the use cases are the *objects* of the concept analysis context, while the executed class methods are the *attributes*. In the resulting concept lattice, the concepts specific of each use case are located (those with extent containing the trace for the given use case only), when existing. Restructuring hints are obtained when the following circumstances hold:

- A use-case specific concept is labeled by computational units (methods) that belong to more than one module (class).
- Different computational units (methods) from a same module (class) label more than one use-case specific concept.

The first case alone is typically not sufficient to identify crosscutting concerns, since it is possible that a given functionality be allocated to several modularization units without being scattered. In fact, it might be decomposed into sub-functionalities, each assigned to a distinct module. It is only when the modules specifically involved in a functionality contribute also to other functionalities that crosscutting is detected, hinting for possible restructuring interventions.

Aspectization of the crosscutting concerns is one of the possible actions to remedy the scattering problem detected by means of feature location. Sometimes, more standard refactoring [4] actions may be sufficient. For example, if a class has too many responsibilities, being involved in several use-cases with specific methods, it might be possible to extract some of its methods/fields, or to build a sub-class out of a part of it (see "move method, move field and extract sub-class" in [4]), thus achieving a better distribution of the responsibilities. When the crosscutting functionality cannot be modularized by means of standard refactoring techniques, AOP is an option.

3.1 Example

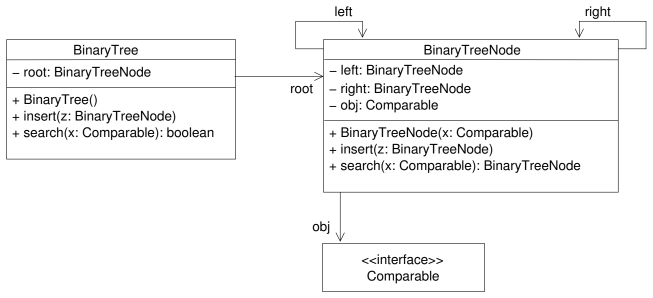


Figure 2. *Classes in a binary search tree application.*

Let us consider a binary search tree application consisting of the two classes depicted in Figure 2. Its main functionalities (to be turned into use cases) are the *insertion* of nodes and the *search* of information inside the tree. Assuming that the second use-case be performed on a pre-loaded tree, the method executions in Table 1 are traced for each use-case.

By applying concept analysis to the relationship in Table 1, the concept lattice in Figure 3 is obtained. It indicates that both the insertion and the search functionalities are crosscutting concerns. In fact, the two use-case specific concepts are labeled by methods that belong to more than one class and each class contributes to more than one functionality. This result can be interpreted as the fact that these two classes are not very cohesive and perform multiple, relatively independent, functions. It would be possible to separate each of these crosscutting functions and

	Insertion
m_1	BinaryTree.BinaryTree()
m_2	BinaryTree.insert(BinaryTreeNode)
m_3	BinaryTreeNode.insert(BinaryTreeNode)
m_4	BinaryTreeNode.BinaryTreeNode(Comparable)
	Search
m_1	BinaryTree.BinaryTree()
m_5	BinaryTree.search(Comparable)
m_6	BinaryTreeNode.search(Comparable)

Table 1. *Relationship between use-cases and executed methods.*

locate them inside a new modularization unit (either a new sub-class or an aspect). For example, a base *BinaryTree* class could be extended by a *BinarySearchTree* sub-class that adds the *Search* functionality to the base tree construction (*Insertion*) functions or a *Search* aspect could be added to the same base *BinaryTree* class.

3.2 Tool support

We implemented our own tool, *Dynamo*[2], to trace method executions of Java programs and we used ToscanaJ[3] for concept lattice construction and visualization. The tracing tool is a modification of the Java compiler *javac*, developed by Sun Microsystems, version 1.4.0. It includes a facility to enable and disable tracing during execution. In this way, it is possible to trace only the portion of execution in which the functionality of interest is actually exercised, skipping any set-up and tear-down phase. With such a facility in place, the assumption that a use-case corresponds to exactly one feature to be located becomes a reasonable one.

4 Case study

Our case study is a Java implementation of the Dijkstra algorithm [2] written by Carla Laffra from Pace University, which supports graph animation during execution. The graphical user interface can be deployed either as a stand alone program or as an applet loaded onto an Internet browser. It is one of the most popular implementations of the Dijkstra algorithm, reported among the topmost entries by Google (searching for "Dijkstra algorithm"). Its migration to AspectJ [10], an AOP extension of the Java language, is described in this section.

A screenshot of this program is shown in Figure 4, while the Lines Of Code (LOC) in the composing classes are

[2]http://star.itc.it/dynamo/
[3]http://toscanaj.sourceforge.net/

	m_1	m_2	m_3	m_4	m_5	m_6
Insertion	×	×	×	×		
Search	×				×	×

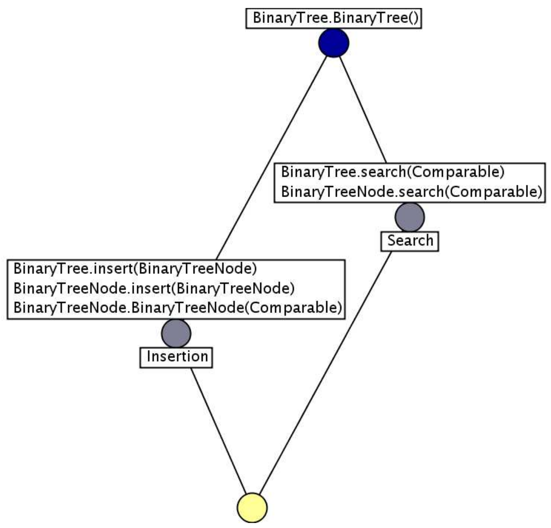

BinaryTree.BinaryTree()
BinaryTree.search(Comparable)
BinaryTreeNode.search(Comparable)
Search
BinaryTree.insert(BinaryTreeNode)
BinaryTreeNode.insert(BinaryTreeNode)
BinaryTreeNode.BinaryTreeNode(Comparable)
Insertion

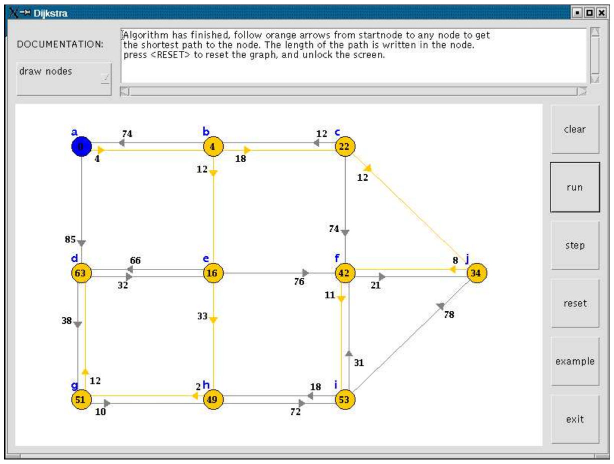
Figure 3. *Concept lattice for the binary search tree application.*

given in Table 2. Although this is a small size application, some features of its implementation makes it an interesting case study.

Class	LOC
DocOptions	35
DocText	106
Documentation	14
GraphAlgorithm	55
GraphCanvas	786
Options	72
Total	**1068**

Table 2. *Size of the classes implementing the Dijkstra algorithm.*

Three use cases have been defined to describe the main functionalities of this program (see Table 3). In the first use case (*Documentation*), the user selects a topic in the topmost-leftmost menu (see Figure 4) and information about the selected topic is displayed in the topmost text area. In the second use case (*Draw*), the user adds, moves and deletes graph nodes and edges. Moreover, some weights are changed, as well as the start node. In the last use case (*Algorithm*) the algorithm is run on a predefined example, both



Algorithm has finished, follow orange arrows from startnode to any node to get
the shortest path to the node. The length of the path is written in the node.
press <RESET> to reset the graph, and unlock the screen.

Figure 4. *User interface of the Dijkstra program.*

in the step-by-step execution mode and in the animated execution mode. The last column of Table 3 reports the number of method executions stored in the traces produced for each test case. Method instrumentation was achieved by means of our tool.

Use case	Description	Execs
Documentation	display documentation	43
Draw	draw graph	2616
Algorithm	run algorithm after loading example	2249

Table 3. *Use cases for the Dijkstra program.*

The context resulting from the execution traces for the three use cases of Table 3 relates 3 objects (the 3 use cases) to 42 attributes (the unique methods executed in some use case). The related concept lattice is reported in Figure 5.

Three use-case specific concepts exist in the lattice, marked by exactly one use case each. Among them, the concept marked by the use case *Algorithm* is regarded as a potential crosscutting concern, since its specific methods belong to three classes (namely, *GraphCanvas, Options, GraphAlgorithm*) with one of them (*GraphCanvas*) appearing in two use-case specific concepts (labeled *Draw* and *Algorithm*).

Among the methods that label the crosscutting concept *Algorithm*, the recurrence of *lock* and *unlock* in all the three classes involved hints for the presence of a candidate aspect. By looking at the bodies of these methods, their function becomes pretty clear. They are devoted to locking a part of the user interface (e.g., node and edge insertion by mouse click, algorithm execution buttons) when the algorithm is running. The other methods labeling this concept handle the execution of the Dijkstra algorithm and belong to the

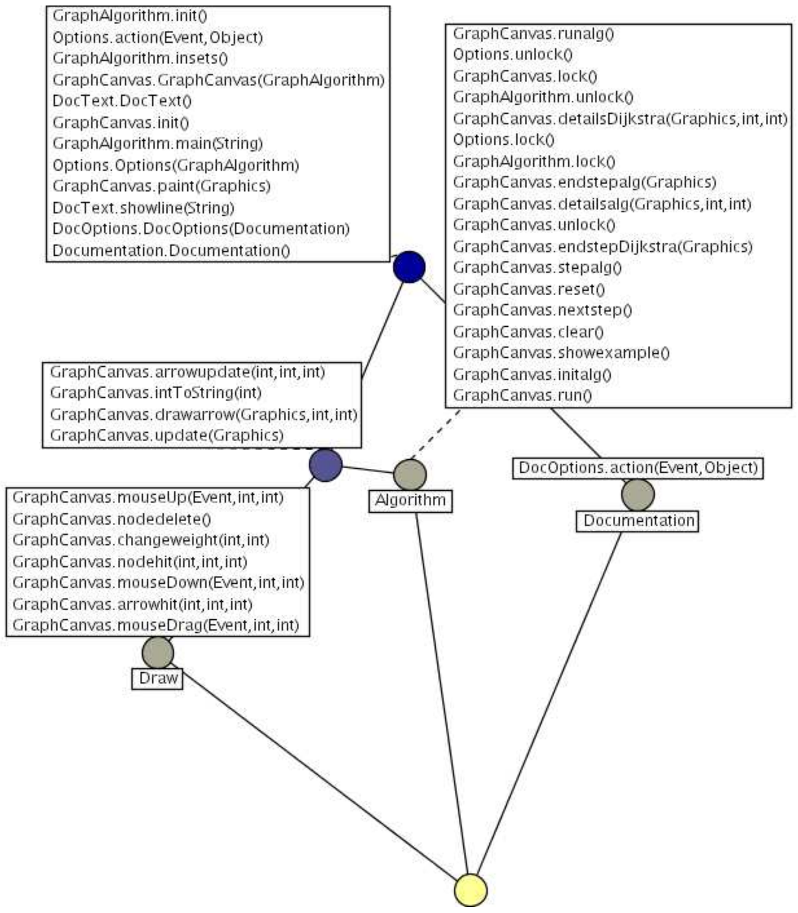

GraphAlgorithm.init()
Options.action(Event,Object)
GraphAlgorithm.insets()
GraphCanvas.GraphCanvas(GraphAlgorithm)
DocText.DocText()
GraphCanvas.init()
GraphAlgorithm.main(String)
Options.Options(GraphAlgorithm)
GraphCanvas.paint(Graphics)
DocText.showline(String)
DocOptions.DocOptions(Documentation)
Documentation.Documentation()
GraphCanvas.runalg()
Options.unlock()
GraphCanvas.lock()
GraphAlgorithm.unlock()
GraphCanvas.detailsDijkstra(Graphics,int,int)
Options.lock()
GraphAlgorithm.lock()
GraphCanvas.endstepalg(Graphics)
GraphCanvas.detailsalg(Graphics,int,int)
GraphCanvas.unlock()
GraphCanvas.endstepDijkstra(Graphics)
GraphCanvas.stepalg()
GraphCanvas.reset()
GraphCanvas.nextstep()
GraphCanvas.clear()
GraphCanvas.showexample()
GraphCanvas.initalg()
GraphCanvas.run()
GraphCanvas.arrowupdate(int,int,int)
GraphCanvas.intToString(int)
GraphCanvas.drawarrow(Graphics,int,int)
GraphCanvas.update(Graphics)
DocOptions.action(Event,Object)
Algorithm
Documentation
GraphCanvas.mouseUp(Event,int,int)
GraphCanvas.nodedelete()
GraphCanvas.changeweight(int,int)
GraphCanvas.nodehit(int,int,int)
GraphCanvas.mouseDown(Event,int,int)
GraphCanvas.arrowhit(int,int,int)
GraphCanvas.mouseDrag(Event,int,int)
Draw

Figure 5. *Concept lattice of the Dijkstra program.*

class *GraphCanvas*.

The first restructuring action that was decided, based upon the inspection of the concept lattice in Figure 5, was the migration of the *Lock* functionality to an aspect. As explained in Section 3, aspects should be both maximally cohesive inside them and decoupled from the principal decomposition. A highly cohesive *Lock* aspect was obtained by moving all the different versions of methods *lock* and *unlock* to the aspect. To increase decoupling from the remaining code, all invocations of such methods were replaced by pointcuts.

Since in AspectJ pointcuts intercept method calls/executions and field access, while it is not possible to intercept an arbitrary statement execution, a preliminary refactoring was required to make the migration possible. Specifically, method *action* of class *Options* handles mouse events on buttons. The code executed for each distinct button is inside an *if*-statement that discriminates the clicked button by name. In some of these code fragments, methods *lock* and *unlock* are called. To replace such direct calls with adviced pointcuts, it was necessary to turn the code fragments for each different button into a separate method ("extract method" refactoring, described in [4]). The resulting code is shown in Figure 6. It should be noticed that such a restructuring is beneficial independently from the migration to AOP. In fact, good programming practice suggests that method *action* should be only responsible for dispatching the mouse events to the handlers, to be implemented as separate methods. Development of a long method *action* with event handling code inlined is usually discouraged.

Figure 7 shows the complete code of the aspect *Lock*. It contains three introductions that add field *Locked* and methods *lock* and *unlock* into class *Options*. Then, it defines the pointcut *lockedActions*, which intercepts the execution of methods *action_step, action_run, action_example* in case the boolean field *Locked* is true. Execution of such methods, associated with the mouse click on some of the user interface buttons, is replaced by the display of an error message in the topmost text area, by means of the following around advice. The aspect *Lock* makes similar introductions into class *GraphCanvas*, where execution of *clear, reset* and *runalg, stepalg* is intercepted respectively by the pointcuts *unlock-*

```
class Options extends Panel {
  ...
  Button b3 = new Button("step");
  private void action_step() {
    b3.setLabel("next step");
    parent.graphcanvas.stepalg();
  }
  private void action_nextStep() {
    parent.graphcanvas.nextstep();
  }
  ...
  public boolean action(Event evt, Object arg) {
    if (evt.target instanceof Button) {
      if (((String)arg).equals("step"))
        action_step();
      if (((String)arg).equals("next step"))
        action_nextStep();
      ...
    }
    return true;
  }
}
```

Figure 6. *Restructured mouse event handling code.*

Class	**LOC**	$\Delta\%$
DocOptions	35	0
DocText	106	0
Documentation	14	0
GraphAlgorithm	45	-18%
GraphCanvas	544	-31%
GraphCanvasExecutor	237	n/a
Options	78	+8%
Lock	82	n/a
Total	**1141**	**+7%**

Table 4. *Size of classes and aspects implementing the Dijkstra algorithm after restructuring.*

GraphCanvas and *lockGraphCanvas*. An after and a before advice are defined respectively to handle the invocation of *unlock* and *lock*. Mouse events on the graph canvas in the user interface are intercepted by the pointcut *lockedMouse*, in turn composed out of three smaller pointcuts. Execution is replaced by the display of an error message by means of an around advice. Finally, methods *lock* and *unlock* are introduced into class *GraphAlgorthm*.

It is interesting to note that migration to AOP allowed for the elimination of some code redundancy. For example, the code in the advice executed around the pointcut *lockedAction* was previously replicated inside all the intercepted executions. A similar situation occurred for all the other pointcuts as well. Thus, previously scattered and replicated code can now be located inside a single, cohesive modularization unit, the aspect *Lock*.

Overall, the size of the migrated application has not changed in a significant way, as apparent from Table 4. Although some code duplications have been removed, the overhead of aspect definition compensated such code shrinking. However, the main expected benefits are not in size reduction, being rather in a better modular structure of the application.

To complete the restructuring suggested by the lattice in Figure 5, the responsibilities of the "fat" class *GraphCanvas* have been reduced by moving those related to algorithm execution to a sub-class, named *GraphCanvasExecutor*. We applied the standard refactoring "extract sub-class" [4]. Inside class *GraphAlgorithm*, creation of an object of type *GraphCanvas* was replaced by the creation of a *GraphCanvasExecutor* object.

The new version of the program was subjected to feature location with the same use-cases as for the original program, resulting in the concept lattice shown in Figure 8. If we look at the use-case specific concepts, it is clear that all crosscutting concerns have been removed. No class contributes to more than one functionality with use-case specific methods.

5 Related works

In the existing literature, aspect mining is based on source code exploration, supported by the outcome of static code analysis [6, 7, 8, 11, 12, 14]. The Aspect Mining Tool AMT, described in [7], supports aspect identification by matching textual patterns against the names used in the code and by looking for repeated uses of the same types. The Aspect Browser tool also uses textual patterns to match the aspects [6]. Their location is improved by adopting a map-based display where aspects are shown in colors. The code browsing tool JQuery is presented in [8]. JQuery provides hierarchical navigation and query facilities, which are useful while executing aspect extraction tasks. Concern graphs [12] can be employed to effectively represent crosscutting concerns. These graphs show the classes, methods and fields involved in a concern and their mutual relationships. They are built incrementally during source code exploration.

Similarly to our paper, in [**?**] dynamic information is used for aspect mining. However, their approach is completely different. In fact, the tool developed by the authors of [**?**] looks for recurring sequences of method invocations that may be turned into aspect advices.

Our paper is the first attempt to let the requirements (translated into executable use-cases) guide aspect identification by means formal concept analysis, applied to the execution traces. The main advantages over the existing approaches is that we do not rely on naming/coding conventions, thus requiring a minimal knowledge about the system under analysis (it is sufficient to define the use-cases for the main functionalitites). Aspects descend from the requirements, thus restoring an alignment with the implementation.

```
aspect Lock {
  boolean Options.Locked = false;

  public void Options.lock() {
    Locked=true;
  }

  public void Options.unlock() {
    Locked=false;
    b3.setLabel("step");
  }

  pointcut lockedActions(Options options):
    (execution(void Options.action_step())
     || execution(void Options.action_run())
     || execution(void Options.action_example()))
    && this(options) && if (options.Locked);

  void around(Options options): lockedActions(options) {
    options.parent.documentation.doctext.showline("locked");
  }

  boolean GraphCanvas.Locked = false;

  public void GraphCanvas.lock() {
    Locked = true;
  }

  public void GraphCanvas.unlock() {
    Locked = false;
  }

  pointcut unlockGraphCanvas(GraphCanvas gc):
    (execution(void GraphCanvas.clear())
    || execution(void GraphCanvas.reset())) && this(gc);

  after(GraphCanvas gc): unlockGraphCanvas(gc) {
    gc.parent.unlock();
    gc.repaint();
  }

  pointcut lockGraphCanvas(GraphCanvas gc):
    (execution(void GraphCanvas.runalg())
    || execution(void GraphCanvas.stepalg())) && this(gc);

  before(GraphCanvas gc): lockGraphCanvas(gc) {
    gc.parent.lock();
  }

  pointcut lockedMouseDown(GraphCanvas gc):
    execution(boolean mouseDown(..)) && this(gc);

  pointcut lockedMouseDrag(GraphCanvas gc):
    execution(boolean mouseDrag(..)) && this(gc) && if (gc.clicked);

  pointcut lockedMouseUp(GraphCanvas gc):
    execution(boolean mouseUp(..)) && this(gc) && if (gc.clicked);

  pointcut lockedMouse(GraphCanvas gc):
    (lockedMouseDown(gc) || lockedMouseDrag(gc) || lockedMouseUp(gc))
    && if (gc.Locked);

  boolean around(GraphCanvas gc):
    lockedMouse(gc) {
    gc.parent.documentation.doctext.showline("locked");
    return true;
  }

  public void GraphAlgorithm.lock() {
    graphcanvas.lock();
    options.lock();
  }

  public void GraphAlgorithm.unlock() {
    graphcanvas.unlock();
    options.unlock();
  }

}
```

Figure 7. *Aspect to lock part of the user interface during algorithm execution.*

In [?] the possibility that aspects address the mismatch between requirements expressed as use cases and implementation is discussed in detail.

Concept analysis was used to restructure existing Object-Oriented code in [?, 13]. Executable subprograms (*concept slices*) for domain specific concepts are extracted in [?]. Their representation inside a concept lattice is the starting point for remodularization in [?]. However, none of these works consider the option of introducing aspects as additional modularization units. Refactoring of existing code toward the Aspect Oriented paradigm was considered in [1, ?, 14]. Recent works in aspect mining based on static source code analysis are presented in [?, ?].

6 Conclusions and future work

The feature location method based on formal concept analysis has been adapted to address the problem of aspect mining. Migration of existing applications to AOP can be supported by our semi-automated aspect identification method, which requires just the definition of use-cases for the main application functionalities, when these are not already available. All the remaining steps, up to concept lattice construction are automated. Interpretation of the concept lattice for migration to AOP is instead a human-intensive activity.

We applied our method to a small case study. The results we obtained are very encouraging, although their generalization to larger programs is hard to make. Thus, our future work we will be devoted to further empirical studies on more realistic examples. However, it is interesting to note that we had no previous knowledge of our case study (chosen randomly among the small size Java programs available on the Web) and that the restructuring hints were obtained only and exclusively by inspecting the concept lattice. The presence of a crosscutting concern (user interface locking) and of a “fat” class with too many responsibilities were apparent from the concept lattice, suggesting the restructuring we performed in a straightforward way. Application of our method to larger case studies might require a few minor changes of the support tool, such as performing an automated analysis of the concept lattice, instead of resorting to its visual inspection, and reporting just the list of candidate aspects detected.

The results presented in this paper are quite promising and represent a provisional validation of the proposed

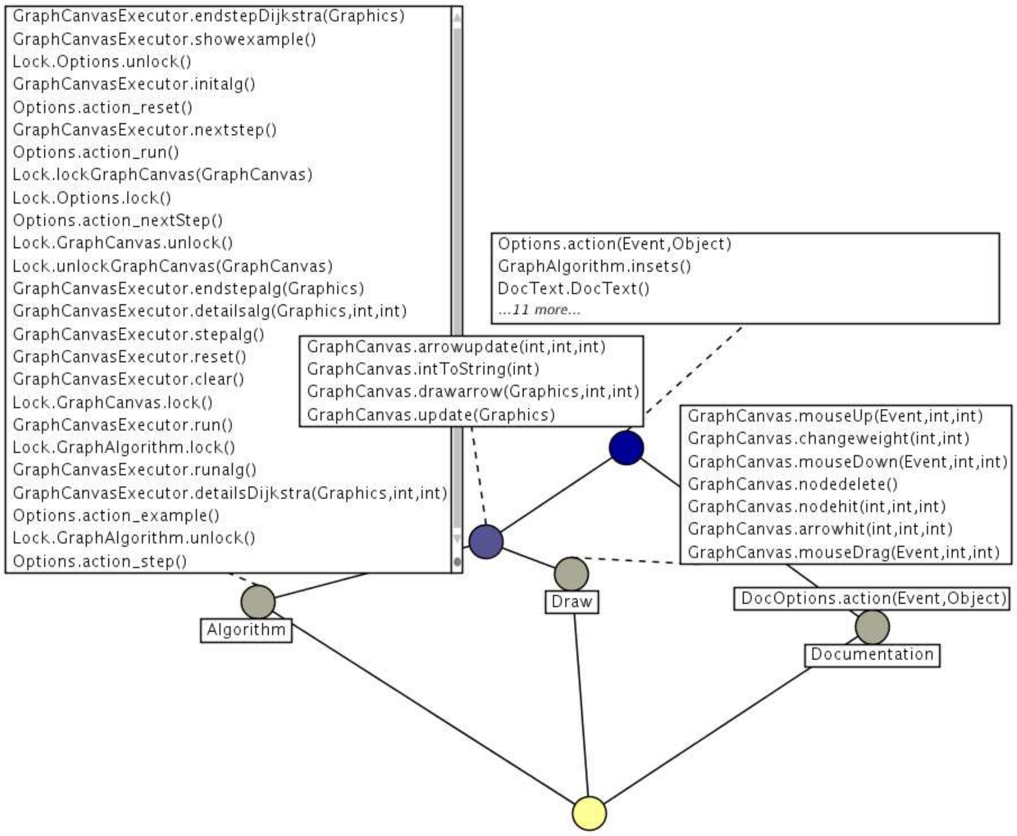

GraphCanvasExecutor.endstepDijkstra(Graphics)
GraphCanvasExecutor.showexample()
Lock.Options.unlock()
GraphCanvasExecutor.initalg()
Options.action_reset()
GraphCanvasExecutor.nextstep()
Options.action_run()
Lock.lockGraphCanvas(GraphCanvas)
Lock.Options.lock()
Options.action_nextStep()
Lock.GraphCanvas.unlock()
Lock.unlockGraphCanvas(GraphCanvas)
GraphCanvasExecutor.endstepalg(Graphics)
GraphCanvasExecutor.detailsalg(Graphics,int,int)
GraphCanvasExecutor.stepalg()
GraphCanvasExecutor.reset()
GraphCanvasExecutor.clear()
Lock.GraphCanvas.lock()
GraphCanvasExecutor.run()
Lock.GraphAlgorithm.lock()
GraphCanvasExecutor.runalg()
GraphCanvasExecutor.detailsDijkstra(Graphics,int,int)
Options.action_example()
Lock.GraphAlgorithm.unlock()
Options.action_step()
Options.action(Event,Object)
GraphAlgorithm.insets()
DocText.DocText()
...11 more...
GraphCanvas.arrowupdate(int,int,int)
GraphCanvas.intToString(int)
GraphCanvas.drawarrow(Graphics,int,int)
GraphCanvas.update(Graphics)
GraphCanvas.mouseUp(Event,int,int)
GraphCanvas.changeweight(int,int)
GraphCanvas.mouseDown(Event,int,int)
GraphCanvas.nodedelete()
GraphCanvas.nodehit(int,int,int)
GraphCanvas.arrowhit(int,int,int)
GraphCanvas.mouseDrag(Event,int,int)
DocOptions.action(Event,Object)
Algorithm
Draw
Documentation

Figure 8. *New concept lattice of the Dijkstra program.*

method. However, several open issues must be considered in the future work. For example, the granularity of the use-cases might affect the quality of the resulting concept lattice, possibly resulting in false positives (too fine grained use-cases) or false negatives (too high-level functionalities). This issue must be investigated in more detail. Moreover, the quality of the aspectized code depends on its internal cohesion, but also on its coupling with the remaining code. Currently, we address the latter issue only by manually defining pointcuts and advices that reduce the dependencies of the remaining code on the aspect. The possibility of automation and of a less subjective assessment of the available options should be also studied in more depth.

References

[1] P. Borba and S. Soares. Refactoring and code generation tools for AspectJ. In *Proc. of the Workshop on Tools for Aspect-Oriented Software Development (with OOPSLA)*, Seattle, Washington, USA, November 2002.

[2] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[3] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):195–209, March 2003.

[4] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley Publishing Company, Reading, MA, 1999.

[5] B. Ganter and R. Wille. *Formal Concept Analysis*. Springer-Verlag, Berlin, Heidelberg, New York, 1996.

[6] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proc. of the 2001 International Conference on Software Engineering (ICSE)*, pages 265–274, Toronto, Canada, March 2001. IEEE Computer Society.

[7] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Proc. of Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE)*, Toronto, Canada, 2001.

[8] D. Janzen and K. D. Volder. Navigating and querying code without getting lost. In *Proc. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 178–187, Boston, Massachusetts, USA, March 2003. ACM press.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *Proc. of the 11th European Conference*

on Object Oriented Programming (ECOOP), vol. 1241 of LNCS, pages 220–242. Springer-Verlag, 1997.

[10] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, Indiana, USA, 2002.

[11] N. Loughran and A. Rashid. Mining aspects. In *Proc. of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (with AOSD)*, Enschede, The Netherlands, April 2002.

[12] M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *Proc. of the 24th International Conference on Software Engineering (ICSE)*, pages 406–416, Orlando, FL, USA, May 2002. ACM press.

[13] G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, 22(3):540–582, May 2000.

[14] A. van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of the 1st International Workshop on Refactoring: Achievements, Challenges, Effects (REFACE), with WCRE*, Waterloo, Canada, November 2003.