Identifying Aspects using Fan-In Analysis

Marius Marin
Software Evolution Research Lab
Delft University of Technology
The Netherlands
A.M.Marin@ewi.tudelft.nl

Arie van Deursen*
Software Evolution Research Lab
CWI & Delft Univ. of Technology
The Netherlands
Arie.van.Deursen@cwi.nl

Leon Moonen
Software Evolution Research Lab
Delft Univ. of Technology & CWI
The Netherlands
Leon.Moonen@computer.org

Abstract

The issues of code scattering and tangling, thus of achieving a better modularity for a system's concerns, are addressed by the paradigm of aspect orientation. Aspect mining is a reverse engineering process that aims at finding crosscutting concerns in existing systems. This paper describes a technique based on determining methods that are called from many different places (and hence have a high fan-in*) to identify candidate aspects in a number of open-source Java systems. The most interesting aspects identified are discussed in detail, which includes several concerns not previously discussed in the aspect-oriented literature. The results show that a significant number of aspects can be recognized using fan-in analysis, and that the technique is suitable for a high degree of automation.*

1. Introduction

Aspect-oriented programming (AOP) is a programming paradigm that addresses *crosscutting concerns*: features of a software system that are hard to isolate, and whose implementation is spread across many different modules. Well-known examples include logging, persistence, and error handling. AOP captures such crosscutting behavior in a new modularization unit, the *aspect*, and offers (compile time) code generation facilities to *weave* aspect code into the rest of the system.

Aspect mining is an upcoming research direction aimed at finding aspect opportunities in existing, non-aspect oriented code. Once candidate aspects have been reverse engineered, they can be used for program understanding or reengineering purposes. Moreover, aspect mining techniques can be used by aspect novices to help them spot opportunities for using AOP techniques. Last but not least, an interesting side effect of aspect mining research is that it increases our understanding of the nature of crosscutting concerns: it forces us to think about what good aspect opportunities are, and it helps us to find aspects that are beyond the canonical ones such as logging and error handling.

Aspect mining research, however, faces a number of challenges:

1. Which techniques can actually help us to find aspects in an automated way?
2. How can we decide that certain functionality should be implemented using an aspect? Since introducing an aspect is a design decision, this is likely to involve a trade off between alternatives. How can we make this trade off and how can we distinguish "good" aspect candidates from "bad" ones?
3. How can we evaluate the quality of an aspect mining technique, in terms of, e.g., precision and recall?

There are no simple answers to these questions. In order to obtain a better understanding of the problem domain, we decided to start out with an, in our opinion, simple and intuitive approach. The analysis we propose involves looking for methods that are called from many different call sites and whose functionality is needed across different methods, classes, and packages. Our approach aims at finding such methods by computing the fan-in metric for each method using the system's static call graph. It relies on the observation that scattered, crosscutting functionality that largely affects the code modularity is likely to generate high fan-in values for key methods implementing this functionality.

We adopt a systematic approach which consists of several steps suitable for a high degree of automation. The technique has turned out to be flexible and easy to combine with other techniques such as clone detection or slicing.

The analysis is focused on three open source systems so that the mined results can be shared and opened to a public debate. We report on the aspects found and explicitly discuss them, including several ones that were not previously investigated in the literature.

The rest of the paper is structured as follows: In the next section we discuss related work. Section 3 describes our approach and illustrates the steps followed for one of the case studies. The most interesting concerns are reported for each

*A. van Deursen received partial support from SENTER, project Ideals, hosted by the Embedded Systems Institute, Eindhoven, The Netherlands.

case study in Sections 4–6. We discuss our results and the future work in Sections 7–8 and conclude in Section 9. More details on the case studies, a list of the selected concerns and a discussion of the methods that led to their identification are contained in the publicly available technical report [17].

2. Related Work

Although the topic of aspect mining is in its early stage of development, it is addressed by several research groups.

The identification of a crosscutting concern typically starts from a so-called *seed*: a method, interface or group of statements part of the concern's implementation. Once seeds have been identified, these can be expanded to full concern implementations by means of manual exploration, slicing, or other techniques. The various aspect mining approaches available to date differ in the techniques used for generating seeds and for extending seeds to full concerns, and in the level of automation provided in these steps.

Shepherd et al. [20] use clone detection based on the program dependence graph and the comparison of individual statement's abstract syntax tree representations, for mining aspects in Java source code. The tool built on this technique aims at a fully automatic mining process and is applied to two of the case-studies considered in this paper. Unfortunately, the list of the mined aspects is not reported.

Two clone detection tools implementing matching on tokens and abstract syntax trees respectively, are evaluated by [3] on an industrial C component. In this component, four dedicated crosscutting concerns were manually identified. The objective is to measure the concern coverage and to assess the suitability of clone detection for reconstructing these annotated aspects automatically.

Interfaces of which implementations are likely to be crosscutting are investigated in [22]. Their technique relies on string matching for the interface name, package membership and call relationships between the methods of the class implementing the interface.

The fan-in of modules was first defined by Henry and Kafura as number of locations from which control is passed into the module (e.g., calls to the module being studied) plus the number of global data accesses [11].

A number of frameworks and tools like Eclipse[1] or FEAT [19] offer code navigation and understanding support, including the determination of a method's callers. Similar results, exposing relationships such as method calls, can be obtained using JQuery[12], an exploration tool with an underlying functional query language. Several mining dedicated tools, like AspectBrowser[8], Aspect Mining Tool [9] and AMTEX[2] , largely rely on lexical analysis - textual pattern matching. The latter two add a complementary type-based mining. All these tools need a starting seed from which the code to be explored further and based on this paper we believe fan-in analysis can provide such seeds. To the authors' knowledge there is no earlier work applying fan-in analysis for generating aspect mining seeds.

The suitability of several techniques, such as clone detection, slicing, dynamic analysis and concept and cluster analysis for the purpose of reverse engineering crosscutting concerns are discussed in [4]. We refer to [3, 4] for more detailed discussions of other aspect mining approaches.

3. Aspect Identification Method

Aspect mining can be conducted in two ways. The first is to take a top down approach, and search for code that implements well-known crosscutting concerns such as logging, persistence, error handling, etc. This calls for a catalog of typical aspects, collected from, e.g., text books or papers discussing aspect solutions to general design problems. Aspect mining research can then provide reconstruction approaches for each of the aspects in the catalog.

The second approach can be considered bottom up, and aims at analyzing the code for the *symptoms* resulting from a lack of proper aspect support in the language used. These symptoms are *code scattering* (the code for one concern is spread across the system) and *code tangling* (in order to implement one concern, the programmer needs to address other (crosscutting) concerns as well). Suitable techniques for identifying such symptoms may be clone detection (for scattering) or slicing (for untangling functionality), as discussed by [4]. Observe that the bottom up approach may be able to discover crosscutting concerns previously not recognized as suitable for an aspect solution.

In practice, aspect mining will be an opportunistic combination of these two approaches. The aspect mining method we propose here also combines both approaches, and originated from an (ongoing) attempt to create an explicit aspect catalog. Aspects in this catalog include generic ones such as logging and profiling [2, 7], as well as exception handling [16]. In addition, many design pattern implementations are candidates for an aspect implementation, as discussed by [10]. Finally, our catalogue contains aspect candidates that are tailored towards business applications such as transaction management, security [15], and J2EE design patterns.

The intuition gained from this study was that many known crosscutting concerns are implemented using one or more methods that exhibit a relatively high fan-in. In terms of AspectJ [2], the method would constitute (part of) the advice, and the call site would provide the context that would need to be captured using a point cut. Based on this observation, we decided to explore in depth how an analysis of methods with a high fan-in could lead us to candidate aspects in a number of open source Java case studies.

[1] `http://www.eclipse.org/`
[2] `http://www.eecg.utoronto.ca/~czhang/amtex/`

```
interface A {
  public void m();
}
class B implements A {
  public void m() {};
}
class C1 extends B {
  public void m() {};
}
class C2 extends B {
  public void m() { super.m();};
}
class D {
  void f1(A a) { a.m(); }
  void f2(B b) { b.m(); }
  void f3(C1 c) { c.m(); }
}
```

Figure 1. Various (polymorphic) method calls

Method	Caller set	Fan-In
A.m	{ D.f1, D.f2, D.f3 }	3
B.m	{ D.f1, D.f2, D.f3, C2.m }	4
C1.m	{ D.f1, D.f2, D.f3 }	3
C2.m	{ D.f1, D.f2 }	2

Figure 2. Fan-in values for code in Figure 1

3.1. The Fan-In Metric

We define the *fan-in* of a method m as the number of distinct method bodies that can invoke m. Because of polymorphism, one method call can affect the fan-in of several other methods. A call to method m contributes to the fan-in of all methods refined by m as well as to all methods that are refining m. Since this metric is fairly simple, except perhaps for the case of polymorphism, we do not provide a formal definition, but illustrate it using an example shown in Figure 1. Three different calls to polymorphic method m are contained in class D. The resulting sets of callers and corresponding *fan-in* values are shown in Figure 2. Observe that the call in f_2 to B's m contributes to the fan-in of m in B's supertypes (A) as well as its subclasses (C_1 and C_2).

We have implemented the *fan-in* metric as a plug-in for the Eclipse Java IDE[3]. The plug-in makes use of standard Eclipse functionality — the *search for references* feature. Our plug-in computes the Fan-In metrics and reports the list of the callers for all the methods in the selected source code — project, package, class, etc. The output allows visualization of the callers and statistical reports.

3.2. Identification Steps

The analysis follows three consecutive steps:

Step 1. Automatic computation of the fan-in metric for all the methods in the targeted source code. The result is stored as a set of "method-callers" structures that can be sorted by fan-in value. This structure can be used to inspect the call sites and calling contexts of selected high fan-in methods.

Step 2. Filtering of the results of the first step:

- Restrict the set of methods to those having a fan-in above a certain threshold. We typically use a threshold of 10, which tends to correspond to around 5% of the total number of methods.
- Filter getters and setters from this restricted set, based on the method's signature, in a first iteration, and its implementation, in a second iteration.

  G(s-)etters on static fields are not eliminated because (as we shall see for example in the PETSTORE case in Section 4) these can be used in the *Singleton* design pattern, which may be a good candidate for an aspect representation [10].
- Filter utility methods, like `toString()`, collections manipulation methods, etc., from the remaining set.

Step 3. (Largely manual) Analysis of the remaining set of methods. The elements considered at this step are the callers and the call sites, the method's name and implementation, and the comments in the source code.

3.3. Example

In this section, we will describe the process of applying these steps to one of the case studies (PETSTORE) to illustrate the systematic approach. In later sections, we will discuss each of the case studies in more detail, putting more emphasis on the aspects that were discovered.

Java Pet Store Demo is a sample J2EE e-business application developed by SUN.[4] It is intended as a demonstration of a real life Web application which allows customers to purchase via a web browser. In addition, it includes modules to perform administration tasks like sales statistics, orders and shipping management, etc.

Pet-store consists of approximately 17,000 non-comment lines of code. Our analysis shows a number of 1855 methods, including the abstract ones. The first step computes the fan-in for each of these methods and distinguishes those having a high fan-in. Considering the results of the other case-studies as well as the ones for the present case, we have decided to analyze the methods with a fan-in value greater than 10, which represents around 1% of the total (1.08% - fig. 3). The chosen reference value, of 10, indicates a significant use of the method in terms of the number of calling methods, and also reduces substantially the number of subjects to be investigated. In a second iteration, we looked at the methods with a fan-in of 8 and 9, which represent another 1.5% of the total. Statistics for the two fan-in sets are shown in Figure 4.

[3] `http://www.eclipse.org/jdt/`

[4] `http://java.sun.com/blueprints/`, PETSTORE version 1.3.2.

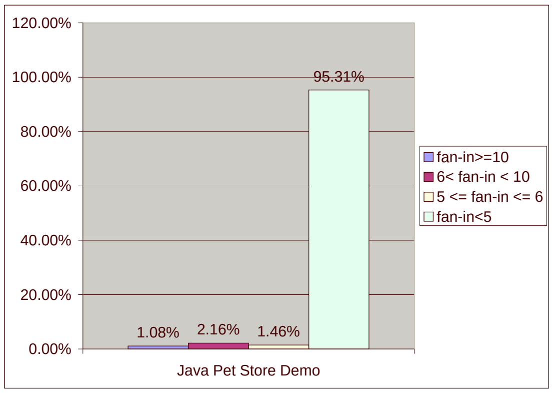

Figure 3. Fan-in distribution for PetStore.

	seeds	non seeds	g(s-)etter	utility	Total
Fan-in∈ [10,∞)					
no.	7	1	7	5	20
%	35.0	5.0	35.0	25.0	100.0
Fan-in∈ [8,10)					
no.	2	16	2	8	28
%	7.1	57.2	7.1	28.6	100.0

Figure 4. High fan-in methods in PetStore

The second step performs a rough filtering of the high fan-in candidates. We were interested in both eliminating those methods likely to provide little information for the last step and in making sure that future automation of the filtering was possible. The filters for g(s-)etters (35%) and utility methods (25%) together eliminated 60% of the candidates from the first step. Most of the getters are still related to aspects to be identified later on. The “utility” subset consist of `toString()` methods and methods belonging to “util” classes, e.g. `XMLDocumentUtils` for building XML documents.

The remaining subset of 40% represents seed (35%) and non-seed (5%) candidates to be processed in the last step. Apart from one case which we decided not to include in the aspects category, all the other methods could be divided into several types of aspects: exception wrapping and business delegates, the Service Locator design pattern, contract enforcement and debugging functionality. All these concerns are discussed in detail in the PetStore section.

Applying the same steps to the methods with a fan-in of 8 and 9, we discovered only two new aspect seeds, one of them related to a new type – the Singleton design pattern. Two other methods, `setup()` and `doTransition(..)`, that occur several times because of the polymorphical mechanism in the search procedure, were considered non-aspects as later discussed in the paper. The 15 instances reported for them cover around 50% of the second set of candidates. The utility methods have largely the same characteristics as the ones already outlined.

```
public class InvoiceTD implements TransitionDelegate {

 /** sets up all the resources that will be needed to do
  *  a transition
  */
 public void setup() throws TransitionException {
  try {
   ServiceLocator serviceLocator = new ServiceLocator();
   qFactory = serviceLocator.getQueueConnectionFactory
                (JNDINames.QUEUE_CONNECTION_FACTORY);
   q = serviceLocator.getQueue
         (JNDINames.CR_MAIL_COMPLETED_ORDER_MDB_QUEUE);
   queueHelper = new QueueHelper(qFactory, q);
  } catch(ServiceLocatorException se) {
   throw new TransitionException(se);
  }
 }

 /** Send an order approval to the OrderApproval Queue...
  */
 public void doTransition(TransitionInfo info)
                         throws TransitionException {
  String xmlCompletedOrder = info.getXMLMessage();
  try {
   queueHelper.sendMessage(xmlCompletedOrder);
  } catch (JMSException je) {
   throw new TransitionException(je);
  }
}}
```

Figure 5. Error handling in PetStore

4. Aspects Candidates in Pet Store

Our analysis of the methods with fan-in above 10 and of 8 and 9 revealed the following cross cutting concerns.

Exception Wrapping and Business Delegates The majority of the aspect seeds are constructors for PetStore exceptions, with fan-in values ranging between 11 and 22. Figure 5 shows the example for the `TransitionException` case, which is thrown from 15 `catch` blocks in different classes and packages.

As in the `InvoiceTD` class in the figure, most of the methods throwing the exception implement `doTransition(..)` and `setup()` declared by the `TransitionDelegate` interface. All the transition delegates handle exceptions related to the particular functionality and re-throw `TransitionException`. This mechanism is in fact part of the *Business Delegate* pattern [1], which aims at hiding the implementation details of a business service. The issue hidden in this case is the sort of exception that can be thrown by the actual implementation.

This mechanism is spread over many places, and a refactoring to aspects is in place. As discussed by Laddad [14], aspects can be used to isolate the exception handling and to wrap the original exception thrown by the underlying implementation in the new exception. This will result in improvements in code size, localization and clarity. Studies of exception handling refactoring [16] show a reduction of `catch` statements when using AOP of up to 95%.

Service locators The method with highest fan-in value (30) belongs to the `ServiceLocator` class, which implements the J2EE pattern of the same name [1]. The intent of the pattern is to provide a single point of control to clients that need to locate and access a component or service in the business or integration tier. The common solution is to have a single instance of the service locator class for an application or, at least, for a tier and thus to have it implemented as a singleton.[5] The delegates previously discussed are part of the locator's clients. They determine a high fan-in value for the locator access methods.

An analysis of the crosscutting nature of the J2EE patterns, including *Service Locator* and *Business Delegate*, was performed by [18]. The authors present a refactoring solution and analyze the advantages of an aspect-oriented implementation.

Contract Enforcement A method with a fan-in value of 27 is a constructor for the `XMLDocumentException` class. This exception is raised if the structure of the XML document does not comply with the expected structure. By examining the call sites, we were able to observe that 9 of them are `fromDOM(Node)` methods, all throwing the exception if a certain compound condition fails. It turns out that all complex conditions share a common check, which can be easily factored out as an aspect by means of before advice – giving rise to the *contract enforcement* concern as discussed in [2]. In this manner, the code will be better localized and new methods will be prevented from omitting the required checks.

Debug Information The `XMLDocumentException` class has a second constructor with a high fan-in. This constructor is (like for the business delegates) used as an exception wrapper. In addition to that, before being wrapped the exception at hand is written on the error output stream. This additional behavior (on top of the wrapping) can be added as another aspect, which indicates which exception should be printed before being wrapped. The aspectization of printing such debugging information is desirable to ensure a common debugging strategy and to isolate the concern that otherwise is crosscutting.

False Alarm The one case considered as non-aspect in the first set of candidates is an `XMLDBHandler` constructor with a fan-in value of 10. The callers are `setup(..)` methods in classes that populate the associated database tables with data from XML files. The `setup(..)` implementations are only slightly different: return an instance of an anonymous inner class extending `XMLDBHandler` that is an XML filter. Because all the callers are well localized in a single package and there is only one `populate(..)` method that triggers the whole process at a client's request, we decided to ignore the candidate.

[5] PETSTORE contains two different service locators, one for JMS and one for EJB components. The former is indeed a singleton (as required in [1]), but the other is not. It does contain a private static field, but this is not used. The fact that this service locator is not implemented as a singleton is an error in the PETSTORE implementation —an error that can be easily fixed by superimposing the AOP singleton solution on top of it.

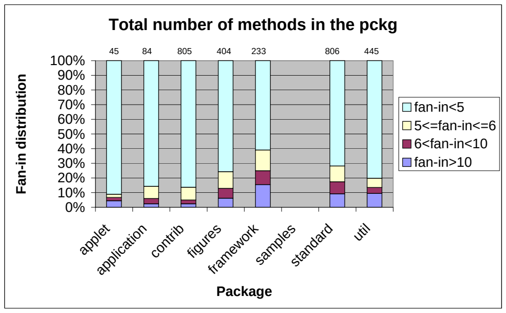

Figure 6. Fan-in statistics for JHOTDRAW.

Lower Fan-In Values The set for the fan-in values of 8 and 9 comprises, because polymorphism, 15 instances of the same two methods, `doTransition(..)` and `setup()`, declared in the `TransitionDelegate` interface. As all transition delegates implement this only interface as their primary concern, the two methods were not considered aspect seeds.

Singleton Two of the interesting methods with fan-in of 8 and 9 are related to the *Singleton* pattern, which is used for one of the `ServiceLocator` classes and the `StatusBar`. The high fan-in method is `getInstance()` which yields the reference to the unique instance. Aspect oriented refactorings for the design patterns from [5] have been proposed by [10] and showed improvements in terms of the modularity properties for most of the cases, including *Singleton*.

5. JHotDraw

JHotDraw[6] is an application framework for two-dimensional graphics. It was designed as an exercise to show a good use of design patterns [5]. The version analyzed is JHotDraw 5.4b1 of approximately 18,000 non-comment lines of code.

Figure 6 shows how the fan-in values for methods are distributed across the various packages of JHOTDRAW. In terms of methods with high fan-in values, above 10, two packages show the top relative and absolute values, respectively: "framework", which contains the classes and the interfaces that define the project's framework, and "standard", which provides the default implementation for the framework classes. The latter is the project's largest package.

[6] `http://jhotdraw.org/`, version 5.4b1

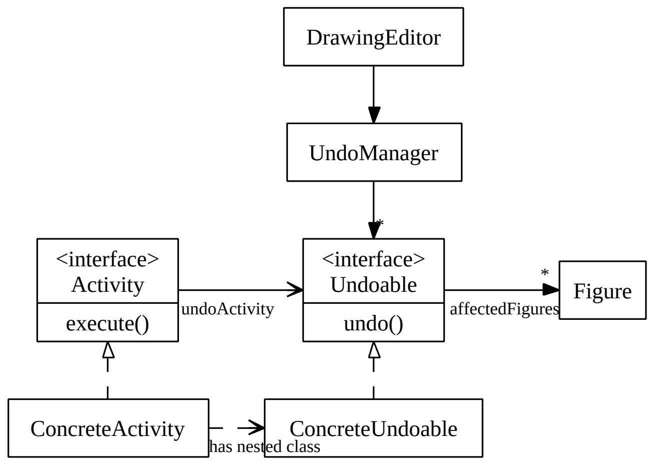

Figure 7. Participants for *undo* in JHOTDRAW.

For each of the two packages, our filters eliminated around half of the methods with top fan-in values. We were rather cautious not to eliminate too many methods. The only methods designated as “utility” are enumerations manipulators (e.g., FigureEnumerator.hasNextFigure()/nextFigure()).

5.1. The Undo Concern

In the top of the list of methods with highest fan-in, there are several methods that contain the word “undo” in their names, or that are part of a class whose name contains “undo”. An undo in a graphical editor is clearly a concern that cuts across many features and activities.

A (somewhat simplified) representation of the participating classes in the JHOTDRAW undo implementation is given in Figure 7. JHOTDRAW offers various sorts of *activities*, which are contained in a class hierarchy. Examples of concrete activities include handling font sizes, triangle rotation, or image rotation.

The interface *Undoable* encapsulates the notion of undoing an action, for which it provides the `undo` method. Each class implementing a concrete activity that can be undone, defines a static nested class conforming to this *Undoable* interface. The nested class knows how to undo the given activity, and has access to all the details of the activity that may be needed for this. Whenever the activity modifies its state, it also updates fields in its associated undo-activity needed to actually perform the undo. In addition to that, a list of *affected figures* is maintained, whose state is also affected if the activity must be undone.

In JHOTDRAW, there are 22 activities that can be undone, causing the undo concern to be spread over these classes. This, in turn, leads to a high fan-in for the methods of, for example, *Undoable*, which helped us to identify this crosscutting concern.

Refactoring undo using aspects takes a number of steps. First, the existing activities can be extended with an association to their undoables by means of a separate *introduction* (through an AspectJ intertype declaration). Second, existing operations should be extended with functionality to keep track of the old state so that the action can be undone. These existing operations can be captured using a pointcut, and then the updates can be contained in advice code. The various nested classes containing the undoable activities can also be added by means of introductions / intertype declarations.[7] The overall effect of this aspectization is that the key functionality remains separated from maintaining the undo state, making both easier to comprehend and modify.

In spite of the fact that “Undo” is a concern that is a natural candidate for an aspect-oriented solution, textbooks on AOP (such as [7, 2, 15]) do not discuss using aspects for undo functionality. On the Codehaus electronic forum, Jon Tirsen provides an interesting generic solution for the undo problem [21]. Tirsen defines a *Command* class, and provides a point cut capturing any field access occurring in the command’s context. He then uses reflection to store the field’s old values in the command, so that a generic undo is made possible.

5.2. Persistence

Another cross cutting concern that pops out clearly through a high fan-in is persistence. Persistence and resurrection of figures is performed by methods inherited from the `Storable` interface. The two are pair concerns: the `Figure` classes implement the methods to write/read themselves to/from a `StorableOutput/Input` object, which is basically a specialized output/input stream. We will discuss only the persistence concern, resurrection being just similar.

Persistence is a crosscutting concern spread over 36 classes. The concrete figures implement the Storable interface that encapsulates the concern. The Storable is what we call a *secondary* interface – an interface that does not define the primary role of the implementing class but only adds some functionality. The methods ensuring the persistent behavior create a high fan-in for several members of the StorableOutput class.

A refactoring solution, using the features of the AspectJ language, would use introduction to isolate the persistence concern from the figure classes. Better modularity properties can be achieved by having all the code relating to the storable functionality in the aspect construct. The figures will implement no persistence-specific code but will be extended with this functionality through inter-type declarations.

5.3. Observers in JHotDraw

The *Observer* is well known as a good candidate for aspect refactoring, due to the roles defined in the pattern’s context

[7] The present version of AspectJ does not yet support introducing inner and static nested classes.

```
public void execute() {
  // perform check whether view() isn’t null.
  super.execute();

  // prepare for undo
  setUndoActivity(createUndoActivity());
  getUndoActivity().
    setAffectedFigures(view().selection());

  // key logic: cut == copy + delete.
  copyFigures(view().selection(),
              view().selectionCount());
  deleteFigures(view().selection());

  // refresh view if necessary.
  view().checkDamage();
}
```

Figure 8. (Simplified) execute method in JHOTDRAW **exhibiting tangling.**

that crosscut the participant classes’ logic. The elements indicative of the *Observer* design pattern are the methods used to *at(de-)tach* the observers to the *subject* participant, to *notify* the *observer* participants of the subject’s changes and to *update* them accordingly. We expect the pattern to be suitable for fan-in recognition through these key-elements. Observers that register themselves as listeners to the subject’s changes increase the metric’s value for the *attach* method. In turn, the *notify* method will be called by every subject’s modifier in order to trigger an *update* operation for the observers.

After these key methods of the *Observer* pattern have been identified by means of fan-in analysis, existing exploration tools and methods can be used to understand the pattern-aspect implementation and its extent. Although the result of this identification method depends on the number of the registered observers and subject changers, our experiments show good results in finding *Observer*’s elements by applying the fan-in metrics.

The set of the methods with high fan-in values comprises a number of seeds for Observer instances such as the `changed()` and `add/removeFigureChange-Listener(..)` methods in `Figure` classes. The value for the `changed()` method is as high as 37. Once defined the Figure’s subject role in the pattern context, an analysis of the call sites will lead to the identification of the other participants.

Matching on the naming conventions used in the identified Observer led to another instance of the pattern (with a somewhat lower fan-in). Thus, fan-in analysis provides initial seeds and application understanding, which then can be used by complementary techniques to identify further cross cutting concerns.

5.4. Other Concerns

JHOTDRAW being an exercise in the use of design patterns, it is not surprising that JHOTDRAW has a high pattern density. Here we analyze which patterns have a cross cutting nature in JHOTDRAW, and how they can be found through fan-in analysis. Hannemann *et al.* provide a general discussion how a number of the classical design patterns could benefit from an AOP implementation [10].

The structure of several design patterns and the associated collaborations can support their fan-in identification. It is yet difficult to turn this identification into a general rule because, most of the time, it is project size and context dependent - whether the design pattern has a key role in the design or just provides an isolated solution.

With high fan-in values reported for *Composites’* children manipulation methods (e.g., `add(Figure)` and `figures()` for standard drawings and thus, for composite figures, and abstract figures) the pattern was readily recognizable.

Singleton classes have been found through the *instance()*-like methods, returning the unique instance of the class (see also Section 4).

Command’s *execute()* operation is invoked by multiple clients: The `execute()` method in the `AbstractCom-mand` class is reported with a fan-in value of 24. There are 10 implementations of this method: An example implementation is provided in Figure 8. The `execute` method is a seed for multiple aspects:

- Each execute implementations starts by checking a common pre-condition, throwing an exception if it does not hold. This is a *contract enforcement* aspect as discussed in [2].

- Most execute implementations conclude with a check if the figure has been changed and a corresponding refresh. This is a *Providing consistent behavior* aspect as discussed in [2].

Factoring these (as well as the undo functionality) out of the code in Figure 8 would leave the execute method with just its core functionality, which is an implementation of the cut operation by means of a copy and a delete operation.

The instances of the *Decorator* and *Adapter* are more difficult to recognize, mainly because unlike the other patterns already discussed, there is not a key method that strongly connotes the link with the pattern. The manual analysis relied more in these cases on the information obtained from naming conventions and the comments in the source code. However, the request forwarding from the decorator/adapter object to the component/adaptee participants could also be considered (e.g., `containsPoint()`, `includes()`, etc. in `DecoratorFigure` all with fan-in values greater than 10).

The *Providing consistent behavior* concern discussed in [2] aims at ensuring that certain functionality is implemented uniformly across a large set of operations. In the `StorableOutput` class, the various printing operations to the *PrintWriter* object are followed by a call to the *space()*

method which simply prints a space to the same object. The method has a high fan-in value, all the calls being from inside the class. It is a good example of crosscutting at class level and also an easy-to-imagine refactoring that will add the *space()* call after the *print(..)*'s execution.

Among the other reported candidates the `displayBox()` method for `Figure` and `Handle` is worth mentioning. It returns the object's display box and is overloaded for figures for changing a figure's display box. The examination of the call sites showed many `draw(Graphics)` callers in handle classes and in similar calling contexts. Despite that, we could not see much benefits from a refactoring. Although the method is reported in several instances, we think it supports the class's main functionality and is not crosscutting.

6. Tomcat/Catalina

TOMCAT is a free, open-source implementation of Java Servlet and JavaServer Pages technologies developed under the Jakarta project at the Apache Software Foundation. Catalina is the servlet container portion of Tomcat.[8] Our analysis is concentrated on the Catalina component. It consists of approximately 90,000 non-commented lines of code.

6.1. Overview of the results

Among the candidates mined, *logging* is one of the crosscutting concerns which is often acknowledged to exist in the Tomcat implementation. The detection of logging by means of fan-in analysis is also not surprising since it is often implemented by means of calls to a commmon method.

The other results found include *context passing* and *contract enforcement* as well as design patterns, such as *Observer*, *Composite*, *Chain of responsibility*, *Command*, *Singleton* or *Adapter*. These candidate aspects all have characteristics similar to those discussed for the other case-studies so we will omit them for conciseness.

The next section describes the new *Lifecycle* concern that we discovered in TOMCAT. We are not aware of any previous discussion of this concern in (AOP) literature.

6.2. Lifecycle

`Lifecycle` is a common interface for the life cycle methods of Catalina components. The documentation states that is intended to provide consistent start and stop mechanisms and should be implemented apart from the interfaces that define the functionality supported by each of the components. This makes Lifecycle a secondary interface — it adds new, supplementary capabilities to the core logic of the implementing classes. In the case investigated, the Lifecycle interface is implemented by about 20 classes.

[8] `http://jakarta.apache.org/tomcat/tomcat-5.0-doc/catalina/docs/api/index.html`, v. 5.0.24

Implementors of the Lifecycle interface are subjects in the context of the *Observer* design pattern and the `start()` and `stop()` methods have to provide the notifying functionality. In addition, these two methods are part of a contract enforcement scheme: The `start()` operation has to be called before any public method of the component, while `stop()` terminates the object's use and should be the last call for a component's instance.

The Lifecycle concern can be seen as a generalization of the use of *stop()* methods to remediate Java's expensive finalization mechanism [23, 6]. Those methods take care of cleaning up the object's resources inside the program code to avoid the overhead of having finalizers but will result in crosscutting for the object's clients.

The Lifecycle concern is complex and although aspect oriented solutions have been presented for some parts of it, a complete refactoring solution remains an open issue. One of the problems with proposing a full solution is that the type of contract enforcement needed by the concern cannot be expressed in a pointcut based aspect language like AspectJ (because it requires specifying "before accessing any public methods of class" and "after last use of class").

7. Discussion

The statistical results show that more than one third of the methods with high fan-in are seeds that lead to aspects. Using filtering, we can automatically remove 60% of the remaining two thirds, leaving almost only seeds of interest.

We can distinguish three situations in which a high fan-in value indicates the presence of crosscutting concerns (in no particular order):

1. The high fan-in method is a key element of the aspect implementation, such as the output method for logging, tracing or debugging functionalities. In such a case, the refactoring to aspects will be driven by the presence of the seed-method calls.

2. The crosscutting implementation is spread over the system and relies on common functionality and the high fan-in method is part of this functionality. In this case, the call sites give the insight into the aspect, e.g., the persistence aspect previously discussed.

3. Some design patterns with a crosscutting structure can lead to high fan-in values when they are given a central role in the project design. This can, for example, happen to the observer manipulation methods that are associated with the subject role in the Observer design pattern. When such methods are found, it strongly implies that the pattern was applied.

We can make a number of observations concerning the results of our fan-in based analysis:

The type of concerns identified - The fact that we were able to find similar aspects in various case studies suggests that their identification is not accidental. The results contains various crosscutting concerns that are discussed in the literature, including those that stood at the origins of AOSD. In addition, we have identified a number of new aspects, such as *Undo* and *Lifecycle*. Given the variety in the case studies, we feel that these results can also be achieved for other cases.

False positives and negatives - We feel that fan-in based analysis is especially good at identifying crosscutting concerns that have a relatively large "footprint" in the source code (since that generally results in higher fan-in). There are both positive and negative sides to this: On the one hand, the aspects identified this way are likely to be those that significantly influence the modularity of the source code and thus most appropriate for refactoring. On the other hand, aspects with a smaller footprint, such as a design pattern providing an isolated solution, could be missed because the number of calls is below the threshold. In any case, the (unfiltered) set of results of fan-in analysis can provide useful information for other mining techniques. Once aware of the potential presence of an aspect, other approaches can be followed to assure or refute it's existence.

It is difficult to discuss false negatives in more detail since it would require a report of all the aspects that could be found in the case studies considered. Such reports are not available at the moment, and we consider our work to be a first step towards a common benchmark for various aspect mining techniques being developed. To promote such a common benchmark, we are in the process of setting up a (wiki-based) web forum[9] where aspect mining researchers can exchange and discuss aspect candidates found in (open source) software systems. We will populate this forum with the results discussed in this paper.

Steps towards automation - The three steps followed in our approach can be automated to a large extend: The first step (Fan-In computation) is already automated as an Eclipse plugin. However, because of the influence of polymorphism during the search for references, in our current implementation the same method could be reported as having high fan-in for different instances. We do not consider this a major issue as it could be repaired by a more in-depth analysis. A more serious limitation could be *fan-in aggregation* — two methods with a relatively high fan-in, but not enough to be individually reported, both calling a third method for which will create a high aggregate fan-in. None of these methods will be reported. An experiment that considers the aggregation would be interesting to perform.

The second (filtering) step is this initial investigation can both be further automated and improved. We generally only filter the simple g(s-)etters (returning or setting a member field value) as they are not likely to offer enough information for further exploration. However, the filtering based on methods' signatures, although substantially reducing the set to be manually investigated, could result in false negatives.

[9] `http://swerl.tudelft.nl/amr/`

The automation of the last step will mainly consist in applying complementary techniques such as clone detection and concept analysis and use their results to both augment and reinforce our own results.

Language independence - An additional case study with fan-in based aspect mining on the 19 KLOC industrial C component that was used in [3] to evaluate clone detection based aspect mining, performed encouragingly well. The main methods in the implementation of the tracing and logging crosscutting concerns have reported fan-in values of 200 and 400, respectively. The metric's value was lower than 10 for the other methods.

8. Future work

One of our goals is achieving a considerable degree of automation. To that end, we are going to extend our mining process as previously discussed. A longer term plan is an aspect mining tool that combines several analysis techniques to achieve a higher degree of completeness and precision.

Fan-in based analysis turns out to be flexible and very suitable for combination with other techniques, such as clone detection or concept analysis. We have just finished a study were we applied the CCFinder [13] clone detection tool to the JHotDraw case study. The clone classes reported in this study as aspect candidates pointed to the same persistence and resurrection aspects identified by fan-in analysis. This result gives us the anecdotal evidence to further explore the combination of these two techniques.

The call site of a high fan-in method can represent a cross cutting concern, but typically the concern is larger than just the call itself, involving setting up appropriate objects and checking relevant conditions. We are investigating how slicing like techniques can be used to find more complete concerns based on initial seeds identified.

Finally, there is a relation that needs to be investigated between the aspectizable interfaces as described in [22] and the methods with a high fan-in value declared by such interfaces. A starting point for such an investigation could be the Undoable interface of JHOTDRAW where a number of methods have reported a high fan-in.

9. Concluding Remarks

In this paper we analyzed three open source Java systems in significant detail, searching for opportunities to use aspect oriented techniques in them. Our search was guided by the fan-in value for each method, following the simple rule that a method with a high fan-in is likely to represent functionality that is required across the application. The key contributions of this paper are the systematic approach used to identify the aspects (described in Section 3) and the discussion of the aspects actually found (in Sections 4–6).

The primary objective of our analysis was to explore the feasibility of reverse engineering aspects from object oriented code. The results of our analysis can be used in the following ways:

- The various examples of candidate aspects can be used to come up with new techniques that can reverse engineer these aspects (semi)automatically.

- The aspects we identified manually can be used in order to validate the effectiveness of new aspect mining techniques.

- The technique can be either used stand-alone or in combination with other techniques.

An interesting side effect of our analysis is that we found a number of opportunities for using aspect oriented techniques, such as the undo and life cycle functionality discovered in JHOTDRAW and TOMCAT, that were not previously described in the aspect-oriented literature.

Last but not least, the nature of the case studies considered provide an assessment that crosscutting is a problem even in the well designed projects. The number of aspects we were able to find enforces the claims that cross cutting is inherent in an OO system. The fan-in analysis, besides of identifying them can give a measure of their impact.

References

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns*. Sun Microsystems, Inc., USA, 2003.

[2] The AspectJ Team. *The AspectJ Programming Guide*. Palo Alto Research Center, 2003. Version 1.2.

[3] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An evaluation of clone detection techniques for identifying cross-cutting concerns. In *Proceedings International Conference on Software Maintenance (ICSM 2004)*. IEEE Computer Society, 2004.

[4] A. van Deursen, M. Marin, and L. Moonen. Aspect mining and refactoring. In *Proceedings of the First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE03)*. University of Waterloo, Canada, 2003.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] B. Goetz. Garbage collection and performance. IBM developersWorks articles, January 2004. `www-136.ibm.com/developerworks/java/`.

[7] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ - Aspect Oriented Programmingin Java*. Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

[8] W. G. Griswold, Y. Kato, and J. J. Yuan. Aspectbrowser: Tool support for managing dispersed aspects. Technical Report CS99-0640, University of California, San Diego, 3, 2000.

[9] J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Proc. Workshop on Advanced Separation of Concerns*. IEEE, 2001.

[10] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 161–173. ACM Press, 2002.

[11] S Henry and K Kafura. Software structure metrics based on information fbw. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[12] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proc. 2nd Int. Conf. on Aspect-Oriented Software Development (AOSD)*, pages 178–187. ACM Press, March 2003.

[13] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfi nder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, 28(7):654–670, 2002.

[14] R. Laddad. Aspect-oriented refactoring. `www.theserverside.com`, December 2003.

[15] R. Laddad. *AspectJ in Action - Practical Aspect Oriented Programming*. Manning Publications Co., 2003.

[16] M. Lippert and C.V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering*, pages 418–427. ACM Press, 2000.

[17] M. Marin, A. van Deursen, and L. Moonen. Identifying aspects using fan-in analysis. Technical Report SEN-R0413, CWI, 2004. `www.cwi.nl/ftp/CWIreports/SEN/SEN-R0413.pdf`.

[18] T. Murali, R. Pawlak, , and H. Younessi. Applying aspect orientation to J2EE business tier patterns. In *Proc. of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*. University of Lancaster, UK, 2004.

[19] M.P. Robillard and G.C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *24th International Conference on Software Engineering (ICSE)*. ACM press, 2002.

[20] D. Shepherd, E. Gibson, and L. Pollock. Design and evaluation of an automated aspect mining tool. In *Proceedings Mid-Atlantic Student Workshop on Programming Languages and Systems*, 2004.

[21] J. Tirsen. Undo in AspectJ. Codehause Jutopia discussion forum, April 25 2004. `blogs.codehause.org`.

[22] P. Tonella and M. Ceccato. Migrating interface implementation to aspect oriented programming. In *Proceedings International Conference on Software Maintenance (ICSM 2004)*. IEEE Computer Society, 2004.

[23] P. Vickers. Why fi nalizers should (and can) be avoided. IBM developersWorks articles, March 25 2002. `www-136.ibm.com/developerworks/java/`.