Improving Aspect Mining with Program Dependencies

By

Navjot Singh

B. Tech. (I.T.), Indian Institute of Information Technology, Allahabad, India, 2004

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University of British Columbia

November, 2006

Abstract

Aspect mining is the process of semi-automatically identifying crosscutting concerns in non-aspect oriented code so that they may be refactored into structured aspect oriented code. In this work, we extend work on aspect mining by examining how patterns of control and data-flow can be used as indicators of aspectual (or crosscutting) behavior. We look for indicators of code which could be refactored into aspects with a clear, narrowly defined interface to the code it would advise. We validated the usefulness of our approach by implementing three analyses and examining the results applied to two open-source projects.

Table of Contents

List of Tables

List of Figures

Acknowledgements

No acknowledgement can sufficiently recognize the inputs that Eric and Kris have made to the work presented here. Thank you, for being such fantastic supervisors!

This research is funded by Microsoft's "Phoenix – Excellence in Programming Award" so a shout-out to them is due here too.

Chapter 1

Introduction

We extend work on aspect mining by examining how patterns of control and data-flow dependencies can be used as indicators of aspectual (or crosscutting) behavior. Our hypothesis is that static code-analysis based aspect-mining techniques can be improved by identifying patterns where fragments of code within a method are controlled or use program data *differently* from one another. Here we consider three patterns which concretely define this notion of differentiation for aspect mining.

Aspect mining is the process of semi-automatically identifying crosscutting concerns in non-aspect oriented code, so that they may be refactored into structured aspect oriented code. The right choice of refactoring can significantly decrease the effort required to understand and maintain large code bases. However, refactoring can be extremely difficult without proper tool support so we seek to improve the state-of-the-art for these tool based approaches.

Mining approaches focus on locating *scattered* or *tangled code* (or both). Code is scattered when logically cohesive fragments are spread across many modules; a method is tangled [14] when logically uncohesive code fragments are interspersed with the primary concern. Fragments might be contiguous statements in a method or statements belonging to a control-flow or data-flow dependency chain. Current mining approaches are based on textual patterns, patterns of method calls, high fan-in methods or duplicated code fragments [1-7]. Our approach is novel in that we consider program dependency information that has not previously been exploited for aspect mining. Although program dependence graphs were used

previously for detecting code clones (scattering), they have not been used for identifying tangled code within methods.

The high-level approach works by looking for distinct interfaces between juxtaposed code in a method and data available in its context. Our intuition was inspired from the work of Walker et al. [16] who claim that aspects should have a clear, narrowly defined interface with the code that they advise. They show that "the separation provided by aspect oriented programming seems most helpful when the interface is narrow (i.e.: the separation is more complete)". Three concrete analyses are presented which are inspired by previous work on program understanding.

The first analysis is primarily based from the work of Rinard et al. who introduce the concept of aspect *scopes*. These scopes are sets of object-oriented class members that are read or written to by class methods or those that are accessed by aspect advice. Different kinds of interactions are classified by examining how the scopes for classes and aspects relate. Unfortunately, since aspect mining works on legacy code, there is no clear distinction between class methods and aspect code. So using the above intuition and taking a reverse approach, *we look for fragments of methods with clearly distinct scopes* to draw a developer's attention to potential refactorings.

The other two analyses are primarily inspired from the work of Ettinger et al. [14] who show how demand-driven program slicing can be used to aid in the extraction of tangled behavior within a class' method. Again applying the above intuition of distinct advice/method interfaces, we compute data dependencies for all statements in a method, *looking for markedly independent data-flows*. We validated the usefulness of our approach by implementing the three analyses and examining the results applied to two open-source projects. The results include a range of code fragments corresponding to behavior widely characterized as

potential crosscutting concerns in aspect-oriented literature. The rest of the report is organized as follows: Chapter 2 presents two examples to motivate the analyses that we implemented, Chapters 3-4 describe the strategies in further detail and present the results observed on 3 different codebases. Chapter 5 discusses some implementation details. Chapter 6 discusses inherent limitations of the approach and builds a case for future research in this area. Related work follows in Chapter 7 and we conclude in Chapter 8.

Chapter 2

Motivating Examples

Now, we discuss examples from real codebases to motivate the approach we have taken.

1. Caching

Significantly different behavior exhibited by two branches of a condition indicate potential refactorings: Conditional branches in the control flow of methods are often points for choosing between alternate behaviors. That leads us to expect that branches will often be the points where crosscutting concerns are being introduced into the primary decomposition. Since all branches are certainly not tangled concerns, we need a more selective strategy. If we could differentiate branches by the behaviors they enclose, the branching points with significantly different behavior on the two branches could be flagged as good advice candidates. The way we differentiate behavior is by keeping track of the state being accessed on a branch using the idea of scopes from Rinard et al. Caching and lazy initialization are two examples of the kinds of concerns we identified using this analysis.

The first example is from the Spring.Net framework (Figure 1), an open-source middleware platform for .NET. This is a typical implementation of object caching. Here, we see that the method `GetNestedObjectWrapper` (line 1)

returns the cached object `nestedwrapper` retrieved on line 9 or creates a new one, adds it to the cache (line 13-23) and then returns it.

According to our hypothesis, the branch instruction at line number 11 that controls whether a new object is created or an existing one retrieved from the cache could be a point of interest for a developer engaging in aspect-oriented refactoring. Our tool can determine that the branch that is taken when the object does not exist in the cache can be differentiated from the one that is taken when it does. This is achieved by looking at how the state of the class is affected on the two paths. Specifically, there is a write to a field variable when the object is not found in the cache but only reads when it does. Differentiating control-flow paths in this fashion makes up our first analysis and is described in detail in Chapter 3.

```
1. ObjectWrapper GetNestedObjectWrapper
2.                 (string nestedProperty)
3. {
4.   /*
5.   * code defining canonicalname
6.   */
7.   // lookup cached sub-ObjectWrapper, create new one
   if not found...
8.
9.   ObjectWrapper nestedWrapper =
   _nestedObjectWrappers[canonicalName];
10.
11.     if(nestedWrapper == null)
12.     {
13.       //Logging
14.       nestedWrapper = new ObjectWrapper (
15.                           propertyValue,
16.                           _nestedPath + canonicalName
   + NestedPropertySeparator);
17.
18.       if(CustomConverters.Keys.Count != 0)
19.       {
20.         //some code to prepare nestedWrapper
21.       }
22.       _nestedObjectWrappers[canonicalName] =
23.          nestedWrapper;
24.     }
25.     else
26.     {
27.       // Logging
28.     }
29.     return nestedWrapper;
30.    }
31.   }
```

Figure 1: Caching in Spring.Net

2. Logging

Tangling [14] can be measured to indicate potential refactorings: A number of well known crosscutting concerns, like logging and failure handling, tend to be

fairly independent of surrounding code. This corresponds to the notion that ideal candidates for aspect refactoring should not be tightly coupled with their context.

A program slice is built on a point of interest in a method and consists of all parts of the method that can potentially affect or be affected by the point of interest. The point of interest - also referred to as the slicing criterion - can be an instruction or an operand.

We claim that some well known crosscutting concerns have limited interactions with their context and that metrics can be used to highlight their presence.

In our second example, we consider a typical implementation of logging. Figure 2 shows parts of a method from Spring.Net, an application framework for the .Net runtime. We are interested in this example because it has the logging concern.

Let us compare program slices built on the logging instructions with those built on the other instructions. A slice built on an instruction in a method includes parts of the method that are related to the instruction by control or data dependencies. The forward slice includes parts that are dependent on the instruction and those on which the instruction depends are included in the backward slice. To keep the discussion short, we consider a slice built on a logging instruction. Notice that the backward slice on 12 shows a dependency on line 5, 4, and 1 transitively. However, the forward slice is empty. Our analysis would detect this as a potential `before` advice as it is tightly coupled to the method input but loosely coupled to the rest of the method body.

In practice, we expect some slices for concerns to be more complicated. Hence, we have devised two ranking schemes (metrics) to rank slices as potential `before` or `around` advice. These are described in Chapter 4. Detection of `after` advice is left for future work.

```
1. object GetPropertyValue(PropertyTokenHolder tokens)
2. {
3.   String propertyName = tokens.CanonicalName;
4.   String actualName = tokens.ActualName;
5.   PropertyInfo pi = GetPropertyInfo(actualName);
6.   if (!pi.CanRead)
7.   {
8.     throw new NotReadablePropertyException(…);
9.   }
10.     if (log.IsDebugEnabled)
11.     {
12.       log.Debug("About to invoke read method [{0}] on
   instance of class [{1}].", pi.Name,
   pi.DeclaringType.FullName) );
13.     }
14.   string keyInCaseOfError = null;
15.   try{
16.     MethodInfo readMethod = pi.GetGetMethod(true);
17.     object val =
   readMethod.Invoke(this.WrappedInstance,null);
18.
19.   //Rest of method omitted
```

Figure 2: Logging in Spring.Net

Chapter 3

Branch Scopes

In Section 2.1 we discussed the intuition for differentiating behavior on different paths taken from a conditional statement. Now we explain the details (Section 3.1) and evaluate the approach (Section 3.2).

1. Approach: Branch Scopes

For every conditional statement in a control flow graph, a branch is defined as the code executed on one of the two outgoing paths up to the point where the control flow meets again or the method returns, whichever occurs first. In this approach two branches are compared based on the properties of their respective scopes. As in [15] we define the scope to be the sets of fields of the class that they read or write to.

Again looking at Figure 1, consider the conditional statement on line 11. One branch comprises line numbers 13-23 and the other comprises line 27. In our analysis, these branches are enhanced in two ways.

We include shared behavior before and after the branches that either affects which branch is taken or is affected by the branch that is taken. To include the code that affects the condition, we build a backward data slice on the guard condition and add it to both branches. Line number 7 and a few others fall in this slice. Next,

forward data dependency slices on the instructions of the two branches are also added. This results in the addition of line number 21.

Having built these new slices, we define what constitutes a significant difference between their scopes. Two slices are considered significantly different when only one of them writes to the state while both may read it. So, we flag the conditional statements where one branch contains a write to a field of the class and the other branch has a read but no write. For instance, in Figure 1, there is a write on line 15 and both slices read the state on line 21.

Table 1: Results for Branch Scopes

Kinds of Results	**Spring. Net Core**	**Log4. Net**
Total	22	67
Caching	7	16
Lazy Initialization	9	19
Other	3	6
False Positives	3	26

2. Evaluation: Branch Scopes

We ran our analysis on two moderately sized codebases. In this section, we summarize the results obtained.

The first codebase we consider is the Spring.Net framework. It is an application framework based on the Spring framework for Java. Spring.Net has many modules and we ran our analyses on Spring.Core (~20K NCLOC). The second codebase is Log4Net (~20K NCLOC), another port of a Java codebase to the .Net runtime. It is a tool to help developers in sending log statements to different output targets.

The analysis successfully identified several instances of crosscutting concerns like caching [9] and lazy initialization [9]. Table 2 shows that a majority of the crosscutting concerns identified fall into one of two categories. We introduce both of these concerns with an example drawn from the results.

a. Caching

Caching, the storing of results from expensive computations for future use, shows up in two of the two codebases. We found 7 occurrences in Spring.Net and 16 in Log4.NET.

Object-oriented implementations for caching vary widely depending on the amount of time and energy expended in designing the caching scheme. Caching has been widely recognized as a crosscutting concern by the AOP community [9]. As discussed in Section 2.1, we expected our analysis to be able to identify caching where it occurs. True to our expectation, a fair number of the results identified are different implementations of caching. To estimate the percentage of

caching code flagged, we searched the source for words like 'cache', 'caching' etc. We found 6 instances of caching using this method and 7 using our analysis. While the keyword search is by no means a precise estimate of all the caching code in the application, our results were a superset of those found by the textual search. This fact, combined with our understanding of common caching strategies and our observation that the Spring codebase is fairly well documented, makes us fairly confident that we are able to flag most, if not all, instances of caching in the codebases. So we conclude that similar caching code occurring in poorly documented or undocumented code would also be identified.

Caching behavior of this kind has been considered a good candidate for aspect oriented refactoring [9]. In the example under consideration, the primary concern of the method is to get the object wrapper from the cache and return it. The tangling with code which deals with creating a new object, registering its type converters and adding it to the cache can be avoided by moving this functionality into the advice of a caching aspect.

b. Lazy Initialization

Lazy initialization refers to the case where some expensive operation such as creation of an object or computing a value is delayed until the first time it is needed. This is another well known crosscutting concern that shows up frequently in our results. We identified 9 cases in Spring and 19 cases in Log4.Net.

Figure 3 shows an instance of lazy initialization identified in the Spring.Net framework. The example belongs to the property ConfigSections in the class PropertyResource Configurer. The property returns the private member variable _configSections if it has a valid value. If it doesn't, it is initialized appropriately and then returned. The analysis identifies the branch at line number 9 as a point of interest. From this point, there is one branch consisting of line number 11 but the other branch is empty (there is no else clause). However, constructing the forward slice (using the technique from Section 3.1) adds line number 13 to both. As a result, we have one branch scope with a write (line number 11) while the other scope only has the read at line 13. This example is a good representative of the other lazy initialization code found by the analysis.

```
1. class PropertyResourceConfigurer
2. {
3.   // details elided
4.   private string[] _configSections;
5.   public string[] ConfigSections
6.   {
7.    get
8.      {
9.        if (_configSections == null ||
   _configSections.Length == 0)
10.          {
11.            _configSections = new string[]
   {DefaultConfigSectionName};
12.          }
13.        return _configSections;
14.        }
15.     }
16.   }
```

Figure 3: Lazy Initialization in Spring.Net

In the example above, the code responsible for initialization is tangled with the primary functionality of the method, which is to simply return the field. In other instances, the primary functionality could be a use of the object being initialized. [9] discusses a recommended aspect oriented refactoring for this concern. The aspect oriented refactoring involves using the get pointcut to advise read access on the field or object and performing the initialization in the advice whenever it is required. An aspect-oriented refactoring is important is this case to prevent programmers from accidentally accessing the field directly and bypassing the lazy initialization code.

c. Other Concerns

This analysis also produced some results from other well known crosscutting concerns such as exception handling (shown in Table 2 under Other). However, we do not report these as positive results for our analysis because such concerns are easily located by searching for keywords in a programming language (e.g. `throws` or `catch`). We don't report these as false positives either because they are easy to filter for exactly the same reason.

```
1. public virtual int Capacity
2. {
3.   set
4.   {
5.      if (value < m_count)
6.      {
7.         value = m_count;
8.       }
9.      if (value != m_array.Length)
10.        {
11.          if (value > 0)
12.          {
13.            IPlugin[] temp = new IPlugin[value];
14.            Array.Copy(m_array, 0, temp, 0,
   m_count);
15.            m_array = temp;
16.          }
17.          else
18.          {
19.             m_array = new
   IPlugin[DEFAULT_CAPACITY];
20.          }
21.        }
22.     }
23. }
```

Figure 4: False Positive due to Backward Slice

d. False Positives

Not surprisingly, our strategy, being quite general, gives some false positive results. Their number is very low on Spring.Net but more significant on Log4.Net. Figure 4 shows a false positive where one of the two branches is empty but a read of a field is introduced into it because we add the backward and forward slices to both branches (as described in Section 3.1). The property `Capacity` in class `PluginCollection` of Log4.Net has an `if` condition on line number 9 with

no `else` block. Notice the backward slice on the `if` condition includes a read of the field `m_count` in line number 7 and this leads to the method being flagged as a result.

We note that it is possible that certain false positives could be considered as application specific crosscutting concerns by a developer more knowledgeable of the semantics for these code bases. Since we had only a surface knowledge, we only report positive results for those that are widely considered as aspects in the literature.

```
1. public void Configure(XmlElement element)
2. {
3. //detail elided
4. LogLog.Debug("XmlHierarchyConfigurator:
   Configuration reset before reading config.");
5.
6. foreach (XmlNode currentNode in element.ChildNodes){
7. if (currentNode.NodeType == XmlNodeType.Element){
8.         XmlElement currentElement =
   (XmlElement)currentNode;
9.         if (currentElement.LocalName == LOGGER_TAG){
10.             ParseLogger(currentElement);
           }
11.      // detail elided
12.     }
13.   }
14.   // Lastly set the hierarchy threshold
15.     string thresholdStr =
   element.GetAttribute(THRESHOLD_ATTR);
16.     LogLog.Debug("XmlHierarchyConfigurator:
   Hierarchy Threshold [" + thresholdStr + "]");
17.     if (thresholdStr.Length > 0 && thresholdStr !=
   "null") {
18.     Level thresholdLevel = (Level)
   ConvertStringTo(typeof(Level), thresholdStr);
19.     if (thresholdLevel != null){
20.       m_hierarchy.Threshold = thresholdLevel;
21.     }
22.   else{
23.     LogLog.Warn("XmlHierarchyConfigurator: Unable
   to set hierarchy threshold using value [" +
   thresholdStr + "] (with acceptable conversion
   types)");
24.     }
25.   }
26.   // Done reading config
27.   }
```

Figure 5: Internal Logging in Log4Net

Chapter 4

Slice Metrics

In Section 2.2 we motivated an approach based on metrics for program slices to capture the interactions with their context. Here, we further refine that discussion with two concrete metrics.

1. Approach: Slice Metrics

We devised two metrics to identify before and around advice candidates based on their interactions with the methods they advise.

a. Around Metric

The first metric was designed for identifying around advice candidates. Logging is a typical example of crosscutting behavior which can be refactored using around advice. We look at an example of logging from log4net and use that to explain our second metric which ranks data dependency slices on methods. Notice that log4net includes logging as *a functional concern and also as a non functional concern* for its own internal debugging by log4net developers. Strategies based on keyword indicators would find it more difficult to distinguish these two behaviors.

Figure 5 shows the method `Configure`. Large parts of the method have been omitted for clarity. Line numbers 4, 16, and 23 are all involved in logging behavior which is tangled with the primary functionality of this method. If we observe the data dependencies between the various logging instructions, we notice

that they do not induce any data dependencies on line numbers 6-13 which belong to the primary concern and are shown in bold. This allows writing logging as an around advice with a clearly defined, narrow interface with the method which executes before and after the method's execution. This suggests a metric on data dependency slices to mine around advice. The metric should "reward" data slices that exclude a significant block of code in the method. The block of code excluded would correspond to the primary concern and hence, should be relatively independent. The data slice that skirts this block of code would correspond to the around advice and hence, would, *ideally*, not intersect with data slices built on other instructions in the method.

We put the above observations together into a metric for identifying around advice. The first step involves constructing forward data slices instead of the program slices constructed earlier. The individual data slices are not representative of the tangling between the slice and the method because they can intersect with forward slices built on other instructions. In Figure 5, the forward data slice built on line number 4 comprises the line numbers 4, 16 and 23. The slice built on line number 15 comprises 15, 16, 17, 18, 19, 20, 23. Neither of the two slices tells the complete story, however. For instance, the first slice does not tell us that line number 23 is also dependant on information from line number 15. Merging the two slices to yield the combined slice is more representative of the data dependencies. Hence, we merge all slices that intersect at any point in the method.

From the set of merged slices, we identify slices that jump over relatively large blocks of contiguous code. Such slices are desirable on account of two factors. First, the relatively large parts of source that are skipped have no data dependencies with the slice. This makes the slice amenable to extraction into advice. Second, a large block of code that is independent of the slice is also more

likely to be the primary concern of the method. The metric, then, boils down to ranking the merged slices by the size of the largest jump. A jump is defined as the number of non -commented lines of source separating two consecutive instructions in the slice.

Table 2: Top Ranked Results for Around Metric

Kinds of Results	**Spring .Net Core**	**Log4Net**
Total	16	30
SecurityContext	0	3
Synchronization	6	19
Logging	1	3
Other	6	0
False Positives	3	5

b. Before Metric

We build slices on every instruction in a method and rank the slices in increasing order of *relative complexity*. First, we provide brief background to relavant concepts in program slicing and then describe our approach.

We compute slices by first constructing a program dependence graph (PDG) [8, 20, 21] for a method and then performing reachability on it. Using a PDG helps

here because the most expensive part of the computation - constructing the PDG - is performed only once. A program dependence graph incorporates both control and data dependence relationships in one graph. Data dependence edges represent the data flow relationships in a program. Control dependence edges are built from the control flow graph and represent the essential control flow relationships in a program. A backward slice, built on an instruction or operand called the slicing criterion, consists of parts of the program that affect the value of the slicing criterion. A forward slice, on the other hand, includes parts that are affected by the slicing criterion. In a PDG with instructions as nodes, the forward or backward slice for an instruction is the set all of all nodes that can be reached in the appropriate direction.

The complexity of slices built on an instruction gives us a measure of how closely coupled the instruction is with the rest of the method. We expect that instructions belonging to the primary decomposition will be closely coupled and have complex slices. To rank slices by complexity, we start with a simple size measure. We count the number of lines of source included in the slice. This is then normalized against the average size of all slices built on that method to give us *relative complexity*. Normalizing against the average size of slices is desirable because it means that slices that are ranked highest are the ones that are most significantly different in complexity from their local context. In other words, we want slices that are small in relation to slices constructed on code surrounding them.

2. Evaluation: Slice Metrics

Table 2 and 3 present the results of computing our metrics on the two codebases. Since we build slices for every instruction in a method, the complete set of results

is actually the entire codebase; programmers are directed to interesting results based on our ranking scheme. The typical use case for this strategy would involve a developer examining results till the number of false positives encountered make further examination non-profitable. This also means that the number of results in each class is not representative of the total number of instances of that concern in the codebase or of the fraction of those concerns identified by our strategy. For our evaluation, we looked at the top few results obtained only. With the Around Metric, we examined results till the gap size was reasonably large. For Spring.Net, this number was 5 while for Log4Net it was 6.

As can be seen from Table 2, the around metric successfully identified 19 instances of synchronization, 3 of logging and 3 uses of the .Net class `SecurityContext in Log4Net`. The number of false positives in the results examined was acceptably low at 5 cases. The results for Spring.Net were similar.

For the before metric, we look at the results obtained in more detail in Section 4.2.

Table 3 summarizes the results obtained on each of these codebases. We were able to identify three classes of widely known crosscutting concerns in the results. We'll discuss some of these results to understand why they are identified.

a. Assertion Checking

One of crosscutting concerns identified by the before metric is assertion checking. This refers to code that throws exceptions or executes special behavior when variables have illegal values or the program is in an illegal state. Again, the aspect oriented refactoring has been widely dealt with in [11]. Figure 6 shows a typical example.

Table 1: Top Ranked Results for Before Metric

Kinds of Results	Spring .Net Core	Log4.Net
Aspects	26	32
Logging	6	7
Assertion Checking	16	15
Other	4	10
False Positives	9	12

Assertion checking is one of the set of systemic aspects that were first conceived as potential use cases for AOP at PARC. Assertion checking involves validating the state of various variables or arguments before computations that depend on that state are performed. Typically, assertion checking code occurs towards the beginning of a method and does not interact much with the rest of it. Due to the small size of slices created, assertion checking was the largest class of results mined.

Figure 6 has some sample code that handles failure conditions in line number 5 through 9. The method `DoAppend` from the class `AppenderSkeleton` sends an error message if an append operation is attempted while the object is in a closed state (lines 5-9). From our previous discussion, it should be obvious that the slices on line number 7 only depends on the class' fields.

```
1. void DoAppend(LoggingEvent loggingEvent)
2. {
3. lock(this)
4. {
5.   if (m_closed)
6.   {
7.     ErrorHandler.Error("Attempted to append to
   closed appender named ["+m_name+"].");
8.       return;
9.    }
10.     //details elided
11.      try
12.      {
13.         m_recursiveGuard = true;
14.         if (FilterEvent(loggingEvent) &&
   PreAppendCheck())
15.         {
16.           this.Append(loggingEvent);
17.         }
18.      }
19.      catch(Exception ex)
20.      {
21.        ErrorHandler.Error("Failed in DoAppend",
   ex);
22.       }
23.      finally
24.      {
25.        m_recursiveGuard = false;
26.      }
27.    }
28.  }
```

Figure 6: Assertion Checking in Log4Net

b. Other Concerns

Besides logging and assertion checking, a number of other concerns came up in smaller numbers. Lazy initialization was one. However, we believe our first strategy was better at finding lazy initialization.

c. False Positives

As seen in Table 3, we encountered a reasonable number of false positives using the before metric. A number of results were due to limitations of our implementation. Mainly, this is because of our current implementation's inability to track data dependencies arising out of the use of Get and Set Properties. For instance, a Set property would induce forward dependencies on the field it sets. Additionally, we don't track side effects of method calls. We've implemented a few work-arounds to mitigate the situation somewhat and these are discussed in Chapter 5.

Chapter 5

Implementation Details

In this Chapter, we provide additional details about our current implementation. The analyses presented are implemented using Microsoft's Phoenix compiler backend. We take .Net binaries as input and raise them to a register-based intermediate representation. We then construct program dependencies from the Phoenix SSA representation.

1. Common Implementation Details

A detail that is common to both our approaches is that our analysis is limited in how it tracks changes to fields of objects. Our current implementation is not able to identify get and set methods with the fields they access. Moreover, side effects of methods are not analyzed; we have not yet implemented a robust inter-procedural analysis. Based on a visual inspection we saw that this led to several of the false positives in both our strategies.

For Brach Scopes, the false positives result when writes to aliases of fields aren't detected as such. With slice metrics, this limitation results in dependencies that are not detected and that gives rise to small slices which, in reality, should be much bigger. We did mitigate the problem by using a few heuristics. First, we parse all methods and flag those that have writes to their fields. While building a slice, if we encounter a call to a method, we look for it in the list of flagged

methods. If it exists, we induce a data dependency on the method receiver. These heuristics are imprecise but can be improved upon with a points-to analysis [18] in the future.

2. Branch Scopes

Our definition of branches excludes sites where loops start. While they are technically points where control flow branches, we didn't feel they were likely to correspond to points that select between two alternate behaviors – a crosscutting concern and a primary concern.

3. Slice Metrics

For slice metrics, we use a measure of slice size that depends on the number of source lines in the slice. Note however, that the analysis works on an intermediate representation. Since there are a number of IR instructions for every line of source, we had to implement a workaround that used the information in program debugging database (pdb) files to find the source corresponding to each IR. When multiple IR instructions correspond to the same source instruction, we use the largest amongst the slices built on these instructions to represent the slice for the line in source.

Another workaround we implemented involved filtering out slices built on initialization instructions. We noticed that slices built on instructions which assigned some default values such as null to local variables would almost invariably be very small since there would be no backward slice and the forward

slice would only extend till the time the variable would be initialized with its proper value. We filtered these results out since they were always false positives.

Chapter 6

Discussion

We have proposed an approach that is significantly different from other work on aspect mining. We validated our intuitions by targeting well known crosscutting concerns. The first half of this Chapter discusses possible threats to the validity of our work. The rest discusses some directions for future work and explains why we feel this is a promising direction for future research in aspect mining.

1. Threats to Validity of Claims

One critique of the work could be directed against the underlying assumption that there is enough information in data-flow patterns to identify crosscutting concerns. In regards to this we think it is important to distinguish between classes of aspects targeting functional crosscutting concerns and non-functional (or systemic) crosscutting concerns. Based on our results and understanding of related work we believe that other work such as [1] which targets design level information (e.g. method naming conventions) will be important for understanding functional concerns. However, systemic concerns such as caching, lazy initialization, may not be logically attached to information in the program design and as such an approach such as ours can be complementary.

Another criticism could be that the results are skewed towards the two codebases chosen. We've tried to address this by discussing the intuitions underlying each

approach. Moreover, the two codebases are all fairly large and that should mitigate some of these concerns.

Finally, our results only consider well known crosscutting concerns. This reflects our desire to confidently validate our results due to the difficulty of verifying application specific refactorings. A solution to this problem would be to analyze a number code bases for which a legacy version and an aspect-oriented version were available. This could serve for a more formal "clean-room" validation of the approach. Unfortunately, we did not have such code available but we expect them to emerge as AOP is further adopted.

2. Future Directions

By targeting well known crosscutting concerns, we've demonstrated that the our hypothesis is partially validated; in one case (caching) we were able to prove low false negatives as well. In all cases the number of false positives was reasonable. In the rest of this section, we discuss some ways to enhance both strategies to target a wider range of aspects.

Branch Scopes: The comparative nature of this strategy currently involves finding pairs of branches where one writes to the class fields and the other only reads. Rinard et al. [15] introduced a classification for aspect interactions including several other types of interaction. For direct interactions, they classify advice based on how and when the method executes after crosscutting. For instance, with *Augmentation* advice the entire body of the method always executes but with *Narrowing* advice, the execution of the method is conditional on the advice. Additionally, they also classify indirect interactions which are based on the fields that are accessed by the advice and method. As we described,

the sets of fields of classes read and written by an advice or method defines it's scope. Depending on the scopes, 5 kinds of interactions are identified. Their classification scheme is more refined than ours but they only use it to classify existing aspects as against mining for aspects in legacy code. In the future, we intend to develop a wider set of branch analyses corresponding based on their classification.

Slice Metrics: The central concept behind out strategies is that good aspects should have a narrow well-defined interface with their context [16]. This led us to the idea to rank slices by a dependency measure. Our current measure for before advice simply measures the size of the slices. However, there are a number of other factors that could be considered when determining the aspect likelihood of a slice. We list some factors that could decrease advice likelihood and explain the intuition briefly:

1. Forward slice includes return value of the method: When methods return a value, any instructions that contribute to that value are not likely to be a part of a before advice.
2. Dependence edges in the forward slice: When an instruction affects computations that follow it in the method, it is more likely to be part of the primary decomposition or an around advice. Hence, our approach could be adaptive to apply the around metric when it detects such a case.

Incorporating the above intuitions and other intuitions about program structure (e.g.: calls to external, static methods are more likely to aspect candidates) will require developing a scoring mechanism to take all the factors into account and could be an interesting direction.

Chapter 7

Related Work

There have been a number of papers that have surveyed aspect mining approaches or compared them. By and large, aspect mining approaches rely on one of the following: textual patterns, patterns in execution, high fan-in methods or duplicated code fragments.

Mens and Tourwe [3] look for textual patterns in method and class identifiers through formal concept analysis. They relying on naming conventions and narrow the results to those that are crosscutting by looking for methods and classes that belong to at least two different class hierarchies.

Breu and Krinke [2] looked for patterns in execution traces to mine aspects. In [7] Breu enhances the approach by using static type information to remove some ambiguities. Tonella and Cecatto [4] perform formal concept analysis on execution traces.

Marin et al [5] find aspects with a large footprint by looking for high fan in methods. Two different techniques for using code duplication to find aspects have been discussed. Shepherd et al [1] look for duplication in the beginning of PDGs. Bruntink et al [6] discuss token-based, AST-based and metrics-based clone detection techniques.

Dependence graphs have been used in software development for optimization [8] as well as refactoring [13]. The use of program slices in refactoring [17] ties in very closely with our first strategy. In fact, Ettinger and Verbaere use slices to untangle crosscutting concerns in methods [14]. Their work is the closest related

work to our first strategy. To the best of our knowledge, there is no prior work on finding aspects by differentiating local interactions in code.

Ishio et al. [24] use a program slicing technique to isolate functional concerns in source code. Different from our approach their approach works from a seed criterion guided by a developer. We feel our approach is complementary in that we identify a class of non-functional concerns indicated by patterns in code.

Many approaches [22-26] involve the developer in the aspect mining process. Our approach can help point the developer in the right direction before using other approaches that bring them into the loop.

Chapter 8

Conclusions

We've laid out the underpinning of our approach to aspect mining based on detection of data-flow patterns. We built on the intuitions inspired by previous work to implement two strategies that targeted well known crosscutting concerns. We tested the strategies on two moderately sized codebases and validated our intuitions. Finally, we discussed important ways in which they can be enhanced to target a wider variety of aspects. The results showed both places for improvement and also a promising approach for future research in aspect mining.

Bibliography

[1] D.Shepherd, E.Gibson, and L.Pollock. Design and evaluation of an automated aspect mining tool. In *Proc. International Conference on Software Engineering Research and Practice*, 2004.

[2] S.Breu and J.Krinke. Aspect mining using event traces. In *Proc. Conference on Automated Software Engineering*, 2004.

[3] K. Mens and T. Tourwe. Delving source code with formal concept analysis. *Elsevier Journal on Computer Languages, Systems & Structures*, 2005. To appear.

[4] P.Tonella and M.Ceccato. Aspect mining through the formal concept analysis of execution traces.In *Proc. Working Conference on Reverse Engineering*, 2004.

[5] M. Marin, A.van Deursen,and L. Moonen. Identifying aspects using fan-in analysis. In *Proc. Working Converence on Reverse Engineering*, 2004.

[6] M. Bruntink, A.van Deursen, R.van Engelen, and T. Tourwe. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proc. of the International Conference on Software Maintenance*, 2004.

[7] S.Breu. Towards hybrid aspect mining: Static extensions to dynamic aspect mining. In *Proc. Workshop on Aspect Reverse Engineering*, 2004.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst., 9(3):319--349, July 1987*

[9] R. Laddad. AspectJ in Action. *Manning Publications Co.,2003*

[10] Filho, F., Rubira, C., Garcia, A., (2005). A Quantitative Study on the Aspectization of Exception Handling. *Workshop on Exception Handling in OO Systems (held with ECOOP), Glasgow, Scotland, 25 July 2005.*

[11] M. Lippert, C. Lopes. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. *In Proc. of ICSE, pages 418--427, 2000.*

[12] M. Weiser. Program slicing. *IEEE Trans. Softw. Eng., 10(4):352--357, July 1984.*

[13] M. Verbaere. Program slicing for refactoring. *MSc thesis, University of Oxford, 2003*

[14] R. Ettinger and M. Verbaere. Untangling: A Slice Extraction Refactoring. *In Proceedings of the Aspect-Oriented Software Development Conference (AOSD), pages 93--101. ACM Press, 2004*

[15] Rinard, M., Salcianu, A., and Bugrara, S. 2004. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth international Symposium on Foundations of Software Engineering*. ACM Press, 2004

[16] R.J. Walker, E.L.A. Baniassad, and G.C. Murphy. An initial assessment of aspect oriented programming. In *Proc. of the International Conference on Software Engineering*, 1998.

[17] W. G. Griswold and D. Notkin. *Automated assistance for program restructuring*. ACM TOSEM, 2(3):228--269, 1993.

[18] R. Ghiya and Laurie Hendren. Putting pointer analysis to work. In *Proc of the Symposium on Principles of Programming Languages*, 1998.

[19] R. Komondoor and S. Horwitz. Effective automatic procedure extraction. In Proceedings of the International Workshop on Program Comprehension, 2003.

[20] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. In Proc. of the Software Engineering Symposium on Practical Software Development Environments, 1984.

[21] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In Proc. of the International Conference on Software Engineering.1992.

[22] C. Zhang and H-A. Jacobsen. Quantifying aspects in middleware platforms. In Proc. of the International Conference on Aspect-oriented Software Development, 2003.

[23] J. Hannenmann, G. Murphy, and G. Kiczales. Role-based refactoring of crosscutting concerns. In Proc. of the International Conference on Aspect-oriented Software Development, 2005.

[24] T. Ishio, R. Niitani, and K. Inoue. Towards locating a functional concern based on a program slicing technique. In Proc. of the Asian Workshop on Aspect-oriented Software Development, 2006.

[25] D. Shepherd, L. Pollack, and K. V-Shanker. Towards Supporting On-demand virtual remodularization using program graphs. In Proc. of the International Conference on Aspect-oriented software development, 2006.

[26] A. Colyer and A. Clement. Large-scale AOSD for Middleware. In Proc. of the International Conference on Aspect-oriented software development, 2004.