This is an incomplete version of the thesis dissertation titled:

Experiences In Migrating An Industrial Application To Aspects

by

Abdelbaset Almasri & Iyad Albayouk

This version of the dissertation does not include some code examples that were drawn from a case study as this case study is covered by a non-disclosure agreement. The full version of this dissertation is available on request. Please contact Andy Kellens or Kris Gybels at Vrije Universiteit Brussel, Dienst PROG, Pleinlaan 2, 1050 Elsene, BELGIUM

Vrije Universiteit Brussel
Faculty of Science
Department of Computer Science

Experiences In Migrating An Industrial Application To Aspects

A Master's Thesis
by
Abdelbaset Almasri & Iyad Albayouk

Promoter: Prof. Dr. Theo D'Hondt
Advisors: Andy Kellens and Kris Gybels

August 2006

Abstract

Aspect-Oriented Software Development (AOSD) is a paradigm aiming to solve problems of object-oriented programming (OOP). With normal OOP it's often unlikely to accomplish fine system modularity due to crosscutting concerns being scattered and tangled throughout the system. AOSD resolves this problem by its capability to crosscut the regular code and as a consequence transfer the crosscutting concerns to a single model called aspect. This thesis describes an experiment on industrial application wherein the effectiveness of aspect-oriented techniques is explained in migration the OOP application into aspects. The experiment goals at first to identify the crosscutting concerns in source code of the industrial application and transform these concerns to a functionally equivalent aspect-oriented version. In addition to presenting experiences gained through the experiment, the thesis aims to provide practical guidance of aspect solutions in a real application.

Acknowledgements

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

We know object-oriented programs or legacy code are structured as a community of interacting objects; therefore most object-oriented programs may have a number of concerns which cannot be localized using the available modularization mechanisms such as persistence, synchronization, exception handling, error management and logging. So these concerns would be scattering and tangling throughout source code yielding what is called crosscutting concerns which make object-oriented programs have several problems arising difficulties in understanding, maintaining and evolving the implementation of the program requirements.

The Aspect-Oriented Software Development (AOSD) is a promising technique that can be considered as one of the most suitable alternatives to improve the software development process of currently legacy systems. AOSD provides valuable additional flexibility in modularization of crosscutting concerns, resulting in considerably better separation of concerns. AOSD does not replace object-oriented programming, it complements it. AOSD improves the modularity of software applications, by extracting the crosscutting concerns in a module called Aspect.

The goal of migration an industrial application from object-oriented to functionally equivalent aspect-oriented version is improving the comprehensibility of the system, and thereby improving its maintainability and extensibility. The migration process could be achieved in two phases: Aspect Mining and Aspect Refactoring.

- *Aspect Mining* can be defined as [***"the activity of discovering those crosscutting concerns that potentially could be turned into aspects, from the source code and/or run-time behavior of a software system"]*** [KM05].
- *Aspect Refactoring* can be defined as ***["the activity of actually transforming the discovered crosscutting concern into real aspects in the source code"]*** [KM05].

1

1.1 Organization of this Thesis

Chapter 2 (Aspect-Oriented Programming): This chapter introduces aspect-oriented programming and the bad symptoms (tangling, scattering) yielding from implementing the crosscutting concern by traditional means of OOP approach. In this chapter we present and explain different AOP languages that provide additional flexibility in modularization and capturing the location and behavior of crosscutting concerns.

Chapter 3 Preliminaries (Aspect Mining, Refactoring, and Java 2 Enterprise Edition (J2EE)): In the aspect mining section, we explain the different aspect mining techniques and discuss how certain of the aspect mining tools can be used. In the refactoring section, we discuss object–oriented refactoring and aspect-oriented refactoring. In this chapter, we give a brief overview of Enterprise Java Beans (EJB), which are used as the underlying technology of the case study we used in our experiment. We also illustrate some of J2EE design patterns, like Service Locator, Value Object, Business Delegate and Session Facade.

Chapter 4 (Aspect Mining in AZ-VUB case study): We describe our experiences applying aspect mining techniques on an industrial legacy application written in Java. We also discuss the aspect mining tools used in this experiment and the crosscutting concerns identified in the application. In the end of the chapter we give an evaluation of the mining activity.

Chapter 5 (Introducing Aspects in AZ-VUB case study): We present in detail the AOP refactoring process applied on the AZ-VUB application. We also discuss and present the refactoring for the crosscutting concerns identified in the AZ-VUB application through the mining process presented in the previous chapter, like: Extracting the Notifying Listener Concern, the Transaction Control Concern, the Exception Handling Concern, the Persistence Concern, the Precondition Checking Concern, the

Exception Wrapping concern and the ServiceLocator Concern. In the end of the chapter we give conclusion of the refactoring process.

Chapter 6 (Road Map): In this chapter, we present the lessons that we have learned through our experiences in migrating an industrial application to aspects. The first lesson outlines the steps and what are involved of the developer effort in extracting the crosscutting concern. The second lesson shows some of the AspectJ limitations. The third lesson explores the refactoring problem of the heterogeneous crosscutting concerns. We also surveyed the steps needed to be followed to migrate from legacy application into aspects. Finally, we explained the pitfalls involved in the migration to aspect.

Chapter 2

Aspect-Oriented Programming

2.1 Crosscutting Concerns

The major ideas in object-oriented programming are build software structure whose behavior reflecting the real-world situation. The live structure of the software being modeled is achieved by describing states and operations that may apply to classes of objects. But many large legacy software systems comprise many concerns that are not localized to a single class; these concerns can be classified into core concerns and system-level concerns. For example, the core concern of an online book shop system would process book orders, while its system-level concerns [java] would handle logging, authentication, transaction integrity, failure recovery, distribution, and so on. Many such concerns are known as crosscutting concerns. The code resulting from implementing these crosscutting concerns will be suffering from a few symptoms. The symptoms can be classified into two categories: [java].

- *Code tangling*: the occurrence of multiple concerns mixed together appears in one module.
- *Code scattering*: the code elements that belong to one concern spread over multiple modules implementing other concerns.

These symptoms make object-oriented software have several difficulties such as: [CCHW04]

1. Difficulty in understanding and reasoning about the implementation of the concern: we must look at multiple areas by the source code for getting the complete picture.

2. Difficulty in adding the implementation of the concern into the code base: Care and attention to detail is required to remind to add logic in each place it must be. Then, at each of these places, the implementation of the concern needs to be done correctly.

3. Difficulty in maintaining and removing the implementation from the code base.

4. Difficulty in reusing the implementation in another system.

Solution

Software developers need an alternative way of thinking about object-oriented program construction. Aspect-Oriented Software Development (AOSD) or called Aspect-Oriented Programming (AOP) provides a new way of thinking about object-oriented program construction and tries to solve problems that confront each developer. AOP provides valuable additional flexibility in modularization to capture the location and behavior of crosscutting concerns, resulting in considerably improved separation of concerns.
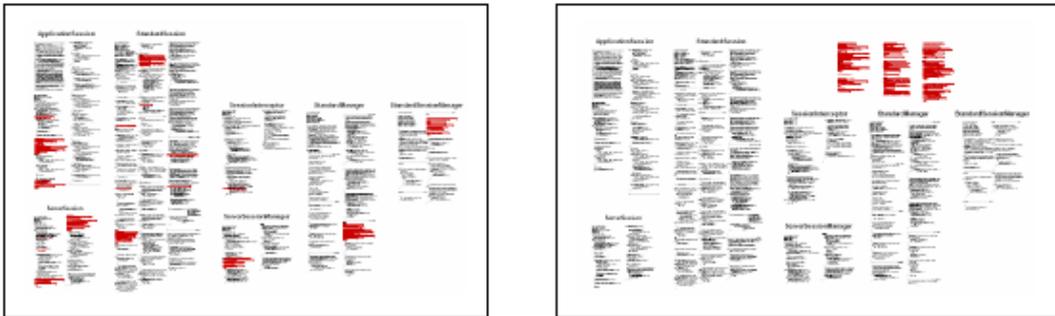
Figure 2.1: Represent Crosscutting Concerns and Aspect Modules

The left side of the figure 2.1 shows the crosscutting concerns leading to tangled code within software code. The code of several modules can be seen in the columns. In those modules the crosscutting code is highlighted. The right side shows AOP addressing this problem by modularizing the crosscutting concerns by means of aspect modules. The software modules are still in place, but the crosscutting code has been extracted and isolated in a single aspect module.

The AOP Approach

AOP is a new way of modularizing crosscutting concerns. Much like object-oriented programming (OOP) is a way of modularizing common concerns. AOP has been proposed as a technique for improving separation of concerns in software. AOP extends object-oriented programming languages by providing modules called *Aspects*. Aspects are for AOP what classes are for OOP. It gathers all the functionality inside of it. It can extend other aspects or classes in the same way as with classes. We can modularize the crosscutting concern in an efficient manner by factoring out logic belonging to the crosscutting concern into an Aspect. The aspects have all the characteristics of the class and add one more. They have potential to enhance the

behavior of other classes through a mechanism called weaving. The process of combining the aspects and the classes into an executable system is called aspect weaving.
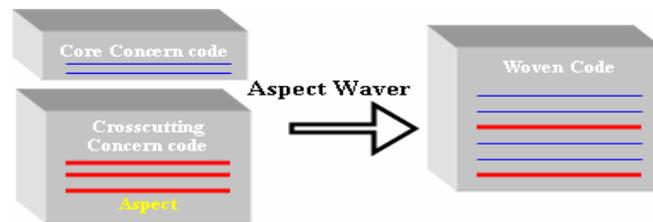
Figure 2.2: Modularizing Crosscutting Concerns

The way of modularizing crosscutting concerns is to separate the crosscutting concern from the core concern and localize it in aspect. Aspect waver is a tool used to combine the crosscutting concern code included in the aspects and the core concern code together yielding a woven code presented the working system. Figure 2.2 visualizes the crosscutting concern modularizing.

Pointcut language is important element of the AO approach. It specifies how an aspect can identify the program's points (join point) where the crosscutting code (aspect code) is joined with core concern code. These joint points could be specified according to behavioral and/or structural properties of the program.

2.2 AOP languages

There are several AOP tools (languages) used to apply AOP approach. [BCC05] AOP languages supply mechanisms that explicitly capture crosscutting structure. These mechanisms make crosscutting concern easily to program in a modular way, and thereby achieve the usual advantages of modularity: easier to understand, maintain and evolve. With aspect modularity, the program has the ability to include / exclude functionality since aspects are separated from the OOP modules, adding or excluding them is a lot easier. Well-known examples of such languages are AspectJ, JAsCo, CARMA, Logic AJ, Alpha, HyperJ, Composition Filters, and CASAR. We will introduce certain of those languages later.

Language Mechanisms used for Capturing Crosscutting Concerns

The primary language mechanisms that AOP languages use to capture and represent crosscutting concerns can be classified as the follows:

Static Introduction

This mechanism is used to handle the concerns known as *static crosscutting concern*. We can alter the program structure by introducing a new operations or fields to existing classes; also we can alter the class hierarchy of the program. Introduction is based on the notion of open classes, and includes addition of fields and methods and declaration of super classes and implemented interfaces. ***Inter-type declarations*** take place at compile time. The introduction mechanism is used by certain AOP languages to handle the many static parts of a crosscutting concern to be expressed in one place, even when the declarations must apply to a variety of separate and unrelated classes.

Affecting Software Behavior Dynamically

We can add extra behaviors at certain points in working system. These points are known as ***join points***. The language's join point model specifies well-defined place in the structure or event in the execution flow of a program at which additional behavior can be added. Join points can be considered as points in a runtime object's life line including points at which the object is created, points at which the object receives a method call and points at which a field of the object is accessed or updated. The join point model may vary considerably between languages. Set of these points can be described by ***Pointcuts*** that is a predicate that matches a set of join points. Join points invoke special code that can alter execution, this code is known as ***Advice***. Through the program execution the advice's code can be triggered at each join point in its pointcut. The implicit advice triggering can be happen:

before the join point: advice can view and modify input values and other state before the join point is entered.

after a join point: advice can view and alter return values and other state after a join point has finished. There are also special cases of after advice for methods returning normally or exiting by throwing an exception.

around a join point: advice replaces the join point. It can view and modify input values, invoke the actual join point using a special keyword, and view and modify its results. It is the only kind of advice that must declare a return type.

2.2.1 AspectJ

AspectJ [asp] is AOP language extension to java language and it is considered most popular AOP language. AspectJ has been developed by team of developers at Xerox's PARC (Palo Alto Research Center). To encourage the growth of the AspectJ technology and community, PARC transferred AspectJ to an openly developed Eclipse project in December 2002. It is the first attempt at a general AOP language. In AspectJ the definition of an aspect is very similar to the definition for the class. The classes contain variables and methods, whereas the aspects contain variables, methods, pointcuts and advices.

Pointcut designators

AspectJ supports several pointcuts designators which can be parameterized with patterns picking out set of methods, constructors, fields and types. The pattern is a regular expression containing "*" wildcard matching any sequence of characters , ".." wildcard in an identifier matching a sequence of tokens starting and ending with "." , and ".." wildcard in parameter list matching any numbers of parameters. For example: *execution(* com..Foo.*(..)   )* matches joint points for execution of any method returning any type in a class Foo of package whose name starts with "com". The method may have any number of arguments.

In AspectJ also we can build compound pointcut by combining the pointcuts using the logical operators and (&&), or (||) , and not (!). For example to capture all calls to methods defined in the java.sql package, or all calls to methods defined in the package javax.sql, we can write the pointcut *call(* java.sql..*(..)) || call(* javax.sql..*(..))*.

The pointcuts designators can be categorized into three categories according to its matching join points; pointcuts designators matching based on join point kind, pointcuts designators matching based on lexical scope of the join point and pointcuts designators matching based on join point context. The two basic pointcut designators from the first category are *call* and *execution*.

- *call(Method-signature)* means a call to method or constructor matching Method-signature, for example *call(public int sum(int,int))*.
- *execution(Method-signature)* means execution of method or constructor matching, for example *execution(public HelloWorld.new(..))*.

The next category of designators that matching join points in specific scope; some of these designators like ***within(type pattern)*** and ***withincode(Method-signature)*** used to delimit the join points according to lexical scope of certain classes or methods ;there are other pointcut designators like ***cflow(Method-signature)*** delimiting join points to be in the control flow of specific method. The pointcut designator ***target(type pattern)*** is an instance of pointcut designators matching based on join point context where the target object is an instance of type matching **type pattern** of the designator.

Advice

Advice declarations can include formal parameters, which are passed to pointcuts and binding values in join points. The body of each advice is executed at every join point captured by the advice's pointcut. [KHH+01, ajd].

Before( ):call(int foo(..)){…} executes before calling the method named foo that returns integer value and takes any number of arguments

before( ):set(int Foo.x) {…} executes before setting a value to the integer field named x in class Foo.

after( )returning:pointcut{…} executes after a normal returning from a join point matched by pointcut.

after( )returning(int x):pointcut{…} executes after a normal returning integer value from a join point matched by pointcut. The variable x is bind to the return value that is accessible to the advice body.

after( )throwing:pointcut{…} executes after throwing any exception in a join point matched by pointcut.

after( )throwing(ExampleException e):pointcut{…} executes after throwing ExampleException in a join point matched by pointcut. The variable e is bind to the thrown exception that is accessible to the advice body.

After( ): pointcut {...} executes after the pointcut, regardless of how it returned whether by normal return or by exception .

String around( ):call(String Foo.toString( )) {... return proceed( );…}: executes instead of calls to toString method of class Foo. The toString () can be invoked in the body using **proceed( )**, which has the same signature as the around advice. The around advice behaves like before and/or after advice, depending on when and if the original join point is invoked.

void around(int nval):call(void Point.set(int))&& args(nval){…}* executes instead of calls to all setter methods of class Point that are parameterized with integer value. The argument value is accessible to advice body.

Reflection at Join points

The AspectJ supports reflective computation on the join point place through a reflective reference that is accessible to advice bodies. Through the special variable *thisJoinPoint* we can access to both the dynamic information at a join point and the static information about the advice: such as the set of arguments at the join point, join point kind (method call, variable read, etc.), signature at the join point, source code location of the join point, object executing the join point, and etc.

Inter-type member declarations

AspectJ support declaration called Inter-type declaration by which we can introduce new elements to other types for instance fields and methods. These declarations are like in form to declarations in those types themselves, except that the member's name is prefaced by a type pattern. The type pattern specifies into which types the member will be introduced. Within the body of introduced methods and constructors, *this* refers to the enclosing object, not to the aspect where the member is declared. For example to introduce the method foo to all classes of type X

public void X.foo( ){//do stuff}

Using the *declare parents* construct; aspects can declare a super class and implemented interfaces on classes. The statement *declares parents: B extends A;* declares that the super class of B is A. Interfaces may be introduced using similar syntax, such as *declare parents: C implements I*; which declares that class C implements interfaces I.

AspectJ Aspect examples

We will explain AspectJ aspect's features by presenting an example of implementing an Observer pattern concern and another example illustrates how implement the checking concern as aspect.

Observer Aspect example:

Code listing 2.1 shows codes of an observer notification concern. In drawing application, when figure elements are moved; the drawing canvas must be notified to

repaint for refreshing its displaying elements. This notifying concern crosscuts all move methods in the figures classes. Every figure class maintains a set of references for its observers by storing or removing these references through special methods to do that. There is also method for notifying the observers after the figure is moved, so we can see the invocation statement of that notify method as last statement in all move methods in figures classes. We need to extract out the logic of this crosscutting concern from the core concern of the figure classes and localize it in an aspect. The aspect in listing 2.1 does exactly this. The aspect introduces the methods manipulating the observer registration and notifying (addObserver, removeObserver, notifyObservers) by using inter-type declarations that appear in the lines 6,7,11 and 15. The aspect specifies when the aspect should be applied by defining pointcut that matches joint points of the move method execution. Also the aspect defines after advice triggered at the pointcut. The action in the advice is notifying the observers of the figure object.

```
1   package aspects;
2   import figures.*;
3   import java.util.*;
4   public privileged aspect ObserverProtocolAspect {
5
6           private Set FigureElement.observers = new HashSet();
7           public void FigureElement.addObserver(Observer o){
8                   this.observers.add(o);
9
10  }
11  public void FigureElement.removeObserver(Observer o) {
12      this.observers.remove(o);
13      }
14
15  public void FigureElement.notifyObservers(){
16          Iterator it = observers.iterator();
17      while(it.hasNext()) {
18        ((Observer)it.next()).update(this);}
19
20          }
21  pointcut moveFigure():execution(void FigureElement.move(int,int));
22  after ():moveFigure(){
23          ((FigureElement)thisJoinPoint.getTarget()).notifyObservers();
24  }
25  }
```

Listing 2.1: Aspect that implement Observer concern for figure classes

Precondition Aspect Example

Precondition checking often requires duplicated code if the conditions are common to many methods. We observe that class Point in listing 2.2 contains two methods setX

and setY checking the parameter value before setting the coordinates of the point with a new value that must be positive value. We can refactor such contract checks into a separate aspect shown in listing 2.3.

```
1  public class Point{
2
3          private int _x;
4          private int _y;
5          public int getX() { return _x; }
6
7          public int getY() { return _y; }
8
9          public void setX(int x) {
10             if(x<0)
11             _x=0;
12         else   _x = x; }
13
14         public void setY(int y) {
15             if(y<0)
16             _y=0;
17         else _y = y; }
18 }
```

Listing 2.2: Class Point including duplicated code for precondition concern in setter methods

Listing 2.2 shows the code without using aspects, and listing 2.3 shows an equivalent program using aspects. By using aspect we can remove this precondition concern from the base code into aspect code.

```
1  public aspect PreCondtionAspect {
2
3      void around( int nval):
4              call(void Point.set*(int))&&
5              args(nval){
6                      if(nval<0)
7                  nval=0;
8                              proceed(nval); }
9          }
```

Listing 2.3: Aspect that implements precondition concern for class Point

2.2.2 JAsCo

JAsCo [jas] is sophisticated aspect-oriented programming language which is designed especially for component based software development (CBSD) [SVJ03]. JAsCo is extension for the Java Beans component model which allows describing reusable aspects, independently from a specific context. The most important features of the JAsCo language are its highest reusable aspects and its strong aspectual composition mechanism to manage combinations of aspects. The JAsCo language is aspect-

oriented extension for Java which as closely as possible to the original syntax and the concepts of Java. JAsCo introduces important additional entities: *aspect bean*, *hook* and *connector*.

- *An aspect* bean allows describing crosscutting behavior in an abstract way, independent of the base application by means of special kind of inner class named hook.

- *A hook* is a structure like AspectJ aspect; it defines advice and part of the pointcut that is independent of the base application.

- *A Connector* is used to deploy aspect beans onto a concrete context and to specify explicit combinations among two or more aspect beans.

JAsCo Aspect examples

Code in listing 2.4 shows a simple JAsCo aspect bean containing a hook. The hook contains around advice that prints a message before and after the execution of the method. Note here that the method captured by the hook is not concrete for any context. The hook constructor takes abstract method signatures as parameters passing them to a pointcut. Code in listing 2.5 explains the hook instantiation using the hook constructor passed to it concrete method signatures used to initialize the pointcut in the hook. This utility can be benefited from it in reuse the aspect to be used for other context. Listing 2.5 shows a connector connecting the tracing hook with all classes of figures package by instantiation the tracing hook with a method signature pattern.

```
1   package  tracing ;
2
3   class AspectTrace {
4           hook Trace {
5                   Trace(method(..args)) { //hook constructor
6                           //metod is absract metod parameter
7                     execution(method); //absract pointcut }
8                   around() {          //advice
9           Tracer.traceEntry("entering "+ thisJoinPoint.getName()+" in "+
10    thisJoinPoint.getClassName());
11                          Object retval= proceed();
12          Tracer.traceExit("Leaving  "+ thisJoinPoint.getName()+" in "+
13    thisJoinPoint.getClassName());
14                  return retval;
15                  }
16          }
17  }
```

Listing 2.4: Simple tracing aspect implementation in JAsCo

```
1   static connector AspectTraceConnector  //connector
2   {
3     application.AspectTrace.Trace hook0 =
4     new  tracing .AspectTrace.Trace(* figuers.Point.*(*)); //hook instantiation
5
6           hook0.around();
7   }
```

Listing 2.5: JAsCo Connector

2.2.3 CaesarJ

CaesarJ [BH05] [cae] is a new aspect-oriented programming language based on Java programming language. CaesarJ language facilitates better modularity and development of reusable components. It provides powerful features, which can be used to improve design of existing Java projects as well for new development CaesarJ has important properties of modularity: abstraction, information hiding and minimization of dependencies. Aspects in CaesarJ are designed as components, which have clear abstraction and can be reusable. CaesarJ improves separation of concern in the same way as AspectJ. AspectJ style pointcuts and advices can be used to intercept points, where component functionality should be integrated. CaesarJ modularizes components, which consist of multiple collaborating classes.

CaesarJ uses the AspectJ weaver, which applies byte code manipulations to insert efficient advice calls. There is no special module construct for aspects in CaesarJ. The pointcuts and pieces of advice are declared directly in Caesar classes. An aspect in CaesarJ is a class, which declares or inherits pointcuts and advice. Aspect objects are instances of such classes. Aspects have all properties of classes: instantiation, encapsulated state, inheritance and polymorphic usage. The inheritance model of CaesarJ is mixin-based. A class can inherit from multiple classes so pointcuts and advices can be inherited from multiple classes.

Listing 2.6 explain an example of an aspect used to trace the execution of all application's methods. As seen in the listing the aspects constructs are defined in normal CaesarJ class.

```
1   public deployed cclass ConsoleTracer {
2   pointcut traceMethods() : (execution(* *.*(..)) ||
3   execution(*.new(..))) && !within(ConsoleTracer+);
4   before() : traceMethods() {
5   System.out.println("Entering [" +
6   thisJoinPointStaticPart.toString() + "]");
7   }
8   after() : traceMethods() {
9   System.out.println("Leaving [" +
10  thisJoinPointStaticPart.toString() + "]");
11  }
12  }
```

Listing 2.6: Simple tracing aspect implementation in CaesarJ [BH05]

2.2.4 CARMA

The CARMA [BH05, kgy] aspect language is a logic pointcut language. The essential language features of CARMA are oriented to the definition of pointcuts. CARMA is an AOP-extension of an object-oriented language; it has a dynamic join point model, very much based on AspectJ's join point model. CARMA's join point model is based on the key events happening in object-oriented programs: sending and receiving of messages, and inspecting and changing of state. At every such event join point, an aspect can intercept and execute advice before or after the actual execution of the join point. The specification of exactly which join points is written in a pointcut language based on logic programming as a logic query over the set of all join points occurring in the object-oriented program. The query can make use of a number of join point predicates, predicates stating conditions over join points, which form the heart of the CARMA language. The most basic predicates are shown in the table 2.1.

Type of join point	Crosscut predicate in old syntax	Crosscut predicate in new syntax (In development)
Message reception	reception(?jp, ?selector, ?arguments)	?jp isReceptionOf: ?selector with: ?arguments
Message send	send(?jp, ?selector, ?arguments)	?jp isSendOf: ?selector with: ?arguments
Assignment	assignment(?jp, ?varName, ?oldValue, ?newValue)	?jp isAssignmentTo: ?varName from: ?oldValue to: ?newValue
Reference	reference(?jp, ?varName, ?value)	?jp isReferenceOf: ?varName havingValue: ?value
Block execution	blockExecution(?jp, ?arguments)	?jp isExecutionOfBlockWith: ?arguments

Table 2.1: Basic crosscut predicates in the CARMA crosscut language for expressing conditions on join points [kgy]

The example in listing 2.7 demonstrates the tracing aspect by defining a pointcuts that capture all reception join points of methods, of all classes in the entire Smalltalk image. The aspect prints a message after and before the captured join point execution.

```
1   before
2   ?jp matching reception(?jp,?selector,?args)
3   do
4   Transcript show: 'Entering ',?selector printString
5
6   after
7   ?jp matching reception(?jp,?selector,?args)
8   do
9   Transcript show: 'Leaving ',?selector printString
```

Listing 2.7: Tracing aspect that traces all method execution in the entire Smalltalk image

2.2.5 Alpha

Alpha [BH05, alp] is an aspect-oriented language with a mostly powerful pointcut model. Pointcuts in Alpha are queries over databases having both static (abstract syntax tree, static type system) and dynamic (full execution trace, heap) information about the program. Alpha supports abstraction mechanisms similar to functional abstraction. This wealthy join point model and the powerful abstraction mechanisms of the pointcut language greatly move up the abstraction level and modularity of pointcuts. Advice is at present as AspectJ (before, after, around). Join point reflection is not needed because necessary information can be passed by means of logic variables from the pointcut within the advice. The example, in listing 2.8, shows five different ways to model a display update pointcut, whereby the lower ones use more semantic information.

```
1    class DisplayUpdate extends Object {
2    Display d;
3    // enum pointcut
4    after set(P, x, _); set(P, y, _); set(P, 'start', _); set(P, 'end', _),
5    instanceof(P, 'FigureElement') { this.d.draw(P); }
6    // set* pointcut
7    after set(P, _, _), instanceof(P, 'FigureElement') { this.d.draw(P); }
8    // pcflow pointcut
9    after now(ID), set(ID, ExpID1, P, F, _), instanceof(P, 'FigureElement'),
10   pcflow(Display, 'drawAll', (_, get((ExpID2, _), F))),
11   hastype(ExpID2, 'FigureElement') { this.d.draw(P); }
12   // cflow pointcut
13   after set(P, F, _), get(T1, _, P, F, _), mostRecent(T2, calls(T2, _, @this.d,'drawAll', _)),
14   cflow(T1, T2), instanceof(P, 'FigureElement') { this.d.draw(P); }
15   // cflowreach pointcut
16   after set(P, F, _), get(T1, _, P, F, _), mostRecent(T2, calls(T2, _, @this.d,'drawAll', _)),
17   cflow(T1, T2), reachable(Q, P), instanceof(Q, 'FigureElement') { this.d.draw(P); }
18   }
```

Listing 2.8: Alpha Aspect [BH05]

2.2.6 HYPER J

Hyper/J [BH05, OT00] is a tool developed at IBM T.J. Watson Research Center. It supports advanced, "multi-dimensional" separation and integration of concerns in standard Java software [BCC05]. Hyper/J is an implementation of the Hyperspaces approach for the Java language. The Hyperspaces approach adapts the principle of multi-dimensional separation of concerns, which involves:

- Multiple, arbitrary dimensions of concern.

- Simultaneous separation along these dimensions.

- The ability to dynamically handle new concerns and new dimensions of concern as they arise throughout the software lifestyle.

- Overlapping and interacting concerns (one might think of many concerns as independent or"orthogonal", but they rarely are in practice).

HyperJ does not use the terms 'join point model' and 'pointcut language' because it is not based on a dominant decomposition approach such as other aspect languages. Instead of expressing an aspect that crosscuts a base program (in a dominant decomposition), HyperJ allows to express multiple decompositions of the program as separate 'hyperslices'. Each decomposition is called a hyperslice. The intention is that each hyperslice contains the implementation of a single concern using the standard programming language constructs (i.e. it is implemented in standard Java). A set of hyperslices can then be merged into a hypermodule using composition rules. The resulting hypermodule implements all concerns implemented in each hyperslice in the composition.

Create hyperspace

As a first step, developers create hyperspaces initially by specifying a set of Java class files that contains the code units that will populate the hyperspace. One way to do this is by creating a hyperspace specification:

```
1   Hyperspace Figures
2   class figurs.*;
3   class Tracer;
```

Listing 2.9: Creation of a hyperspace

Hyper/J will automatically create a hyperspace with one dimension – the class file dimension. A dimension of concern is a set of concerns that are disjoint. The initial hyperspace will contain all units (interfaces, classes, methods, and member variables) in the corresponding class files within the specified package.

Create concern mappings:

To create a new dimension (Feature dimension) can specify concern mappings, which describe how existing units in the hyperspace address concerns in that dimension:

```
1   package figures: Feature.Kernel
2   class    Tracer:  Feature.Tracing
```
Listing 2.10: Concern mappings

The first line indicates that, by default, all units contained within the figures package address the Kernel concern of the Feature dimension. The second line specifies another mapping indicating that **class** named "Tracer"   address the tracing concern.

Create hypermodules:

By means of hypermodule specifications one can define hypermodules, which are modules based on concerns. A hyperspace can contain several hypermodules realizing different modularizations of the same units. Systems can be composed in many ways from these hypermodules. In this hypermodule, the Kernel and tracing concern are related by a "mergeByName" integration relationship. This means that units in the different concerns correspond when they have the same names ("ByName") and that corresponding units are to be combined; for example, all members in similar classes are merged into one class.

```
1   hypermodule Figures_With_tracing
2   hyperslices: Feature.Kernel, Feature.Tracing
3   relationships: mergeByName;
4   merge Feature.Tracing.Tracer with *;
5   bracket *  with
6   Feature.Tracing.Tracer.traceEntry(ClassName, MethodName)
7   Feature.Tracing.Tracer.traceExit(ClassName, MethodName)
8   end hypermodule;
```
Listing 2.11: Hypermodule specifications

The "merge" relationship expands on the "mergeByName" relationship; it indicates that the Tracer Class unit in the tracing concern from the Feature dimension is to be merged with all other class units in the other hyperslices, even though their names differ. The "bracket" relationship indicates that all methods should be *bracketed* by the methods *Tracer.traceEntry* and *Tracer.traceExit*. Thus, for example, each *move*( ) method in the composed hyperslice will call *Tracer.traceEntry* upon entry and *Tracer.traceExit* before exit. The parameters passed to these bracketing methods will be the names of the class and method, to identify the method called. The bracket relationship is very useful when we need to add behavior to the beginning and/or end of methods.

2.2.7 Composition Filters

Composition Filters (CF) [com, BA04] is approach developed at the TRESE group, at the Department of Computer Science of the University of Twente, The Netherlands. CF approach is an extension of the object-oriented programming. The primary idea behind CF is that messages that received by OOP object can be intercepted, and manipulated in various ways, modifying the form in which the object behaves. To do so, in the CF model, a layer called the interface part is introduced.

Filter type	Accept Action	Reject action
Dispatch	The message is dispatched to the specified target of the message	The message continues to the next filter in the set.
Error	The message continues to the next filter in the set.	An exception is thrown
Wait	The message continues to the next filter.	The message is queued while the evaluation of the filter expression results false
Meta	The reified message is sent as a parameter of another –meta message- to a named object. The object that receives the meta message can observe and manipulate the message, then reactivate its execution.	The message continues to the next filter in the set
Substitute	certain properties of the message can be substitute	The message continues to the next filter.

Table 2.2: Filter types and the taken actions when the message is accepted or rejected

The primary components in the CF model are the input filters and output filters. Each type of these filters implements a particular manipulation of messages. The filters together compose the behavior of the object, possibly in terms of other objects. After the composition of filter modules and filters, received messages must pass through the input filters, and send messages through the output filters.

All filters have a common structure; a name that specifies the filter, the type of the filter and a set of expressions that define the way of messages filtering. There is a behavior attached for each type of filter to identify the actions taken when the filter accepts or rejects the messages matching the pattern defined in the filter. Some predefined filter types are show in the table 2.2.

2.3 Aspect language comparison

In this section we present the primary elements of the aspect languages discussed above. So we discuss the language properties of join point model and pointcut language. The main element of each aspect language is the join point model that describes the points where additional behavior is attached. The join point models can be identified by the following properties:

1. **Dynamic (AspectJ-based) join points:** The join points are matching points that can be captured in the execution of the program.
2. **Static join points:** The join points are static program elements.

The pointcut language is another element of an aspect language. It specifies how an aspect can identify the join points. The pointcut language can be characterized by the following properties:

3. **Logic query language:** The pointcut language is a logic query language.
4. **Behavioral properties:** The pointcut language allows describing the join points based on the behavioral (dynamic) properties of the program.
5. **Structural properties:** The pointcut language allows describing the join points based on the structural (static) properties of the program.
6. **Pattern-based pattern:** The pointcut language allows describing the join points using regular expressions.
7. **(AspectJ-based) predicates:** The pointcut language includes a set of predicates that can restrict the join points.

Table 2.3 summarizes the above properties for each aspect language discussed in the previous section.

language	Join point model		Pointcut language properties				
	Property 1	property 2	property 3	property 4	property 5	property 6	property 7
Alpha	x		x	x	x	x	x
AspectJ	x			x	x	x	x
CaesarJ	x			x	x	x	x
CARMA	x		x	x	x	x	
JAscO	x			x	x	x	x
HyperJ		x			x		

Table 2.3: Aspect language properties

2.4 Summary

This chapter introduced aspect-oriented programming. We identified the bad symptoms (tangling, scattering) yielding from implementing the crosscutting concern by traditional means of OOP approach. We have known how AOP approach mechanisms can clear software code from these symptoms yielding maintainable software. We have seen different AOP languages that provide additional flexibility in modularization to capture the location and behavior of crosscutting concerns, resulting in greatly improved separation of concerns.

Chapter 3

Preliminaries (Aspect Mining, Refactoring, and Java 2 platform, Enterprise Edition (J2EE))

3.1 Aspect Mining

Software developers try to improve object-oriented programs (legacy system) using aspects, because the object-oriented programs may have many concerns which cannot be localized using the available modularization mechanisms. So these concerns would be scattered and tangled throughout the source code yielding what is called crosscutting concerns which make object-oriented programs very difficult to understand, maintain, and reuse. Consequently, software developers need tools and techniques for aiding them to detect those crosscutting concerns in legacy system. The activity of detecting the crosscutting concerns in a legacy system is called aspect mining. Nowadays there are several aspect mining tools and techniques that can be classified into two kinds: dedicated browsers and automated aspect mining techniques.

In this section we give an overview of the different aspect mining techniques. We also give discuss how certain of the aspect mining tools can be used.

3.1.1 Dedicated browsers

Dedicated browsers require a starting point (also called seed) of a concern to manually identify those crosscutting concerns by discovering the legacy system. Dedicated browsers may have a query language to aid developers for searching for crosscutting concerns [KM05]. Dedicated browsers have a number of advantages and disadvantages: the advantage is that the developers can identify exactly the concerns they want, in exactly as much detail as they need. The disadvantage, of course, is that much of the cognitive burden is placed on the developer, with the tool acting more as a recorder than a helper and developers need a seed of a concern to search manually for those crosscutting concerns in legacy code [HT05]. There are several examples of dedicated browsers like:

3.1.1.1 The Feature Exploration and Analysis Tool (FEAT):

FEAT is developed as a plugin for the Eclipse Platform [RM02]. FEAT represents concerns as a tree in Concern Graph. A Concern Graph is for saving a set of concerns related to a particular task. A concern is for saving program elements (classes, methods, and fields) of interests and the relations between themselves. FEAT allows developers to search, browse, understand, and analyze the code implementing a concern in a Java system. By visually navigating structural program dependencies, developer can determine the code implementing a concern, and save the result as an abstract representation consisting of building blocks that are simple to manipulate and query. The representation of a concern supported by FEAT can be used to explore the relationships between the captured concern and the base code, and between the different parts of the concern itself. FEAT has three main views, see figure 3.1:

- *The Concern Graph View,* displays the hierarchy of concerns for a given Concern Graph.
- *The Participant View,* displays all the program elements and their relations which are concerned in the concern selected in the Concern Graphs View.
- *The Projection View,* displays query results.

 Developers can find each seed of concerns by using manually searching in Package Explorer of Eclipse or using automated aspect mining tools, when a concern of interest is identified, it can be modeled with FEAT. To do so it is necessary to create a Concern Graph. A Concern Graph can represent several concerns all linked to a task. Once a Concern Graph is created, it is possible to either add program elements to the current concern in the Participants View, or to query an element in the Projection View. Elements can be queried or added to a concern or projection through the context menu either in the Eclipse Package Explorer or Outline View. Concerns can also be compared.

Model of FEAT

Concern Graph [RoMu02] is a subset of a structural program model built by FEAT. The program model represents the declaration and uses of different program elements of class-based object-oriented languages. Formally, a program is expressed as a graph $P = (V, E)$, where V is the set of vertices, and E is the set of labeled, directed edges.

A vertex in *P* can be one of three types.

- **Class vertex (I)** represents a global class or interface, without its members.
- **Field vertex (F)** represents a field member of a class.
- **Method vertex (M)** represents a method member of a class.

An edge in *P* can be one of six types, depending on the type of vertices it connects: (M, M), (M, F), (M, C), (C, C), (C, M), and (C, F). Edges are labeled with the semantic relationships they represent. A number of examples of edges that connect vertices of *P* are shown in table 3.1.

Name	Type	Description
(**calls**, *m1*, *m2*)	(M, M)	The body of method *m1* contains a call that can bind (dynamically or statically) to method *m2*.
(**reads**, *m*, *f*)	(M, F)	The body of method *m* contains an instruction that reads a value from field *f*.
(**writes**, m, *f*)	(M, F)	The body of method *m* contains an instruction that writes a value from field *f*.
(**checks**, *m*, *c*)	(M, C)	The body of method *m* checks the class of an object, or casts an object, to *c*.
(**creates**, *m*, *c*)	(M, C)	The body of method *m* creates an object of class *c*.
(**declares**, *c*, {*f*|*m*})	(C, F|M)	Class *c* declares method *m* or declares field *f*.
(**superclass**, *c1*, *c2*)	(C, C)	Class *c2* is the superclass of *c1*.

Table 3.1: A number of relationships in FEAT [RoMu02]

For example, if a class called A has a method called *m*(int, int), there will be an edge from class A to method *m*(int, int) called *declares*.

In FEAT, an aspect is defined as a subset of the graph *P* documenting the implementation of a concern in *P*, and it is stored in a structure called Concern Graph. FEAT gives a set of queries to allow developers to access vertices of the program model that are associated to the vertices in the Concern Graph. A developer can navigate the program model in both the direct and reverse directions of the edges dribbling from the vertices.

There are two groups of queries in FEAT:

- Fan-in: returns all the vertices in the program model that depend on the selected class, field or method node.
- Fan-out: returns all the outgoings edges for the selected node. Fields don't have outgoing edges.

See table 3.2 and table 3.3 for describe the queries we can do in FEAT.

FEAT has a numbers of advantages, like:

- The main advantage of Concern Graphs is to use them to save our information as we explore different concerns of importance in a program.

- The developer can fast determine and analyze concerns scattered in an existing code base.

- The key concept of comparing two concerns is observe how they be linked without having to understand the whole concern.

- Concern Graphs could be extended to extra programming languages, including procedural languages such as C.

FEAT has a numbers of disadvantages, like:

- The developer implements the relations defined and queries as static by using the FEAT.

- The developer can't add new queries to explore new types of feature relations.

- The developer needs to be customary with Eclipse Platform.

- The developer needs starting point of concerns to start analysis the code.
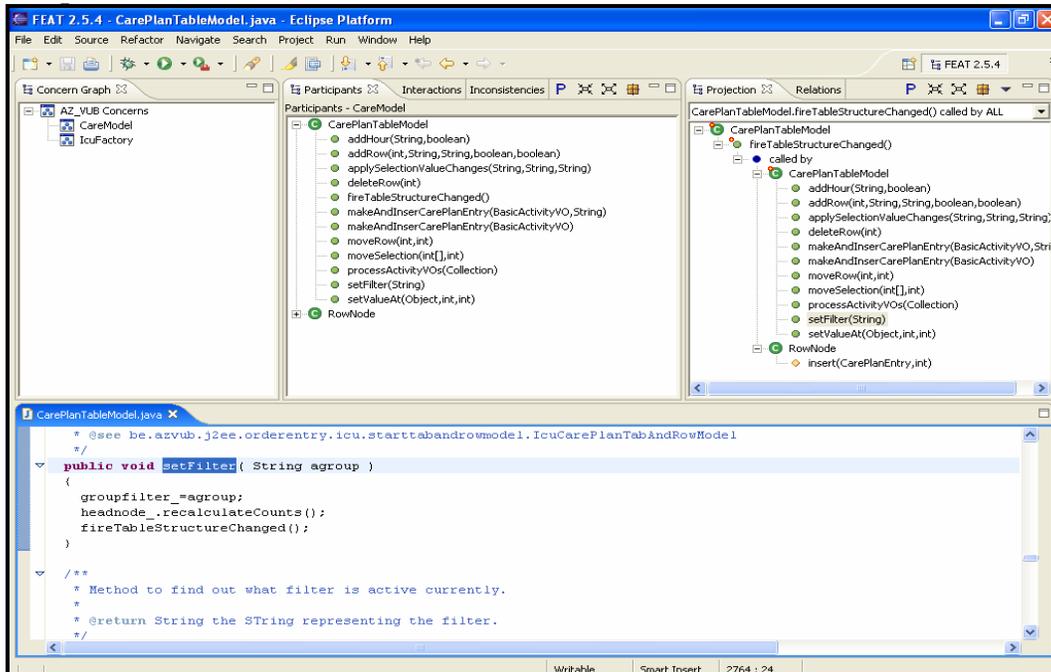
Figure 3.1: FEAT Perspective

25

Query Name	Applicable to	Returns
declaring	Classes	All the members of the class
extending	Classes	The direct superclass
i-extending	Interfaces	The direct superinterface
implementing	Classes	The interfaces the class implements
transitively extending	Classes	All the direct and indirect superclasses
transitively implementing	Classes	All the interfaces implemented by this class, directly or indirectly
being of type	Fields	The type of the field, if non-primitive
creating	Methods	The classes of objects created in the body of the method
having p-types	Methods	The non-primitives parameter types of the method
having r-type	Methods	The return type of the method, if non-primitive or void
accessing	Methods	The fields accessed in the body of the method
calling	Methods	The methods called, including methods potentiall resulting from dynamic binding
overriding	Methods	The methods that this method overrides
using	Methods	The fields used, object created, and methods called in the body of the method

Table 3.2: FEAT Queries (Fan-out)

Query Name	Applicable to	Returns
created-by	Classes	All the methods creating an object of the class
extended-by	Classes	The direct subclasses
i-extended-by	Interfaces	The direct subinterfaces
implemented-by	Interfaces	The classes that directly implement this interface
transitively extended by	Classes	All the direct and indirect subclasses
transitively implemented by	Interfaces	All the classes implementing by this class, directly or indirectly
accessed by	Fields	All the methods accessing the field
called by	Methods	All the methods calling this method, including methods which might call it through dynamic binding

Table 3.3: FEAT Queries (Fan-in) (part 1)

Query Name	Applicable to	Returns
overriden by	Methods	All the methods that override this method
referenced by	All	All the classes/methods/fields that relates to the queried object

Table 3.3: FEAT Queries (Fan-in) (part 2)

3.1.1.2 Aspect Browser

Aspect browser is developed as a plugin for the Eclipse Platform. Aspect browser for Eclipse allows developers to visualize programs in a Seesoft-like view by searching for regular expressions and displaying the results graphically. Additionally, aspect browser includes features to navigate through search results and manage a potentially large set of regular expressions [AB].

SeeSoft [ESS92] is mainly employed to visualize the files based by text such as the source code. It traces each row of text into a line with the color indicating statistics of interest. The statistics can be any attributes derived for the source, such as the history of revision or the frequency of execution. The main advantage of SeeSoft is that it can clearly reduce the size of the representation thus of the interesting visual patterns can be found and these patterns are often connected to the attributes which are repeated in the data.

The goal of aspect browser is to aid developers to display, explore, and handle crosscutting concerns. So all the files in a program are displayed as a row of small windows in which each line of code in a file corresponds to a row of pixels in a window. Each occurrence of a crosscut is highlighted in a window with a specific color, like symbols on a map see figure 3.2.

Aspect browser has two main views [AB]:

- *Aspect Tree View*: In the Aspect Tree View we can create and edit aspects and manage them into groups. In addition, we can view computed source information that performs a lexical analysis of our programs and shows all existing Eclipse markers.

- *Visualization and Navigation View:* The Visualization and Navigation View offers a graphical "map" of your packages and the files in each package. From this high-level view we can determine how modularized or crosscutting an aspect is.
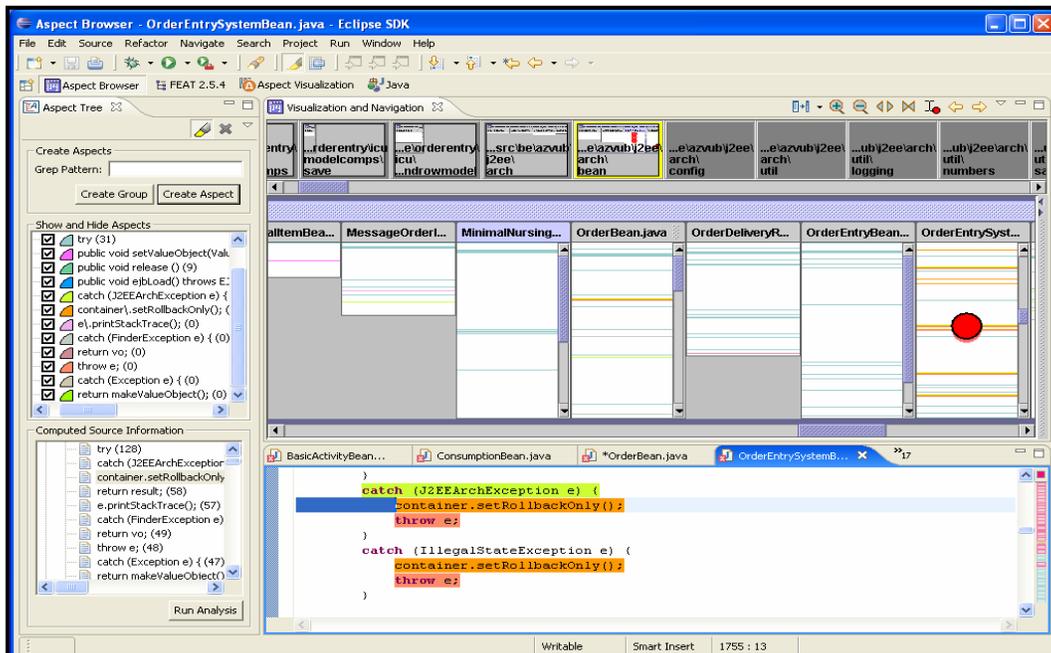
Figure 3.2: Aspect Browser Perspective

Aspect browser has a numbers of advantages, like:

- Aspect browser is a graphical tool that aids developer to find and manage aspects.
- Aspect browser gives the developer a quick understanding how a crosscut is dispersed across the files.
- Aspect browser has features which aid a developer to find possible representatives of crosscutting concerns, such as the identification of redundant lines of code.

Aspect browser has a numbers of disadvantages, like:

- Developer perhaps can't view too a lot of aspects at a time because of the overwhelming number of colors. Also, on a larger project, the number of aspects will increase, and an approach to arrange aspects will be essential.
- Aspect browser only achieves textual-pattern searches; it doesn't differentiate between a package name, a type name, a variable name, a method name, or a code comment.
- The developer needs a lot of time to analyze and to filter the results.

28

- The developer needs starting point of concerns to start analysis the code.

3.1.1.3 Aspect Mining Tool

The Aspect Mining Tool (AMT), developed by Jan Hannemann, provides an open multi-modal analysis framework for concern identification and system understanding. AMT offers two analysis techniques to search for possible Aspects [amt]:

- *Lexical (text-based) Analysis*: This offers simple pattern matching same as aspect browser.
- *Type-Based Analysis*: With type-based analysis, code tangling can be detected and modularity quality measures as coherence and coupling of the code can be visualized.

The Aspect Mining Tool consists of two rather independent programs [amt]:

- The *analyzer* extracts all necessary line-oriented program statistics (currently: source code and types used) and structural information (currently: package and class hierarchy information). All extracted information is written to a data file.
- The *visualizer* uses the data file to display a line-based view of the system (for example, compilation units as collections of lines of code). Developers can then query the system database (created by the visualizer from the data file) interactively.

AMT has a numbers of advantages, like:

- The AMT provides an open multi-modal analysis framework for concern identification and system understanding.
- The type-based analysis works pretty well with objects and variable.

AMT has a numbers of disadvantages, like:

- The AMT works finest if naming conventions for types, methods, variables and classes are followed. The code that doesn't follow such naming conventions is not detected.
- The type-based analysis doesn't work with method invocations. The tool doesn't discover the signatures of method invocations; they have to be detected with textual searches.

- The AMT is not possible to build a concern data structure in order to store different query results.

- The AMT is quite old and the results are not associated to the source code, making the tool nearly useless.

3.1.1.4 Prism

Prism is developed as a plugin for the Eclipse Platform see figure 3.3, the aspect mining activities in Prism are centered on three main concepts [ZJ04]:

- *Fingerprints*: In Prism, a fingerprint is a representation of a certain trait of an aspect or a particular coding concern. A basic fingerprint provides a direct description of the coding pattern. A composite fingerprint provides an abstract pattern definition which is a Boolean combination of any other fingerprints. Composite fingerprints express more complex traits through the reuse of already defined fingerprints. The current Prism implementation supports binary AND and OR expressions through operators && and ||. Currently, Prism supports three different categories of coding patterns. The simplest patterns are lexical patterns in the program texts using regular expressions. Prism also supports lexical patterns on type names and method names as well as patterns of inheritance relationships. Moreover, Prism supports any valid Java code fragment for representing call of methods. Each Prism fingerprint is associated with two types of filters in making search results more specific. Scope filters use either namespace information, for example, package names in Java systems, or regular expressions on type names to cover the entire code base or any of its subsets. Lexical filters can be used to specify the lexical patterns of the actual text of the code. Lexical filters are used in conjunction with fingerprints specified using type patterns so that patterns of both type names and their instance names can be captured. Prism provides GUI based fingerprint builders and facilitates the lifecycle management of fingerprints.

- *Advisors*: Prism advisors are tools, each of which autonomously computes an independent characteristic of the code base in order to assist precise definitions of fingerprints for aspects. While the most desired feature of an advisor is the

automatic discovery of convoluted concerns, a powerful advisor can make good suggestions of possible convolutions and their possible locations in the code. Based on this information, a fingerprint can be defined to accurately capture the code level representation of these properties. Currently, Prism provides a ranking advisor which reports most frequently-used types across methods.

- *Footprints*: Footprints are matches of fingerprints in the code base. They are the results of the queries represented as Prism fingerprints. The current implementation of footprints is able to represent matches at the granularity of lines. Matches of lexical patterns and call patterns are individual lines in the source code.

Prism has a numbers of advantages, like:

- Provides a large variety of ways for developer to describe an aspect through prism fingerprint definitions.
- Enables search of calling patterns defined at package level, class level, and method level. So supports the AspectJ call pattern convention.
- Supports navigation between mining results and source locations.
- Provides automatic discovery of aspects for developer through ranking advisor.
- Supports quantification of kind usage scattering during computing degree of scattering and scattering ranking.

Prism has a numbers of disadvantages, like:

- Not support Mining of multiple languages.
- Not contain facilities to determine relationship between program elements.
- Prism does not achieve a super-type matching on the method's declaring-type and on each of its arguments. For example: assume we have the following type definitions:

```
interface A{
  public void m( );
}

class B implements A{
  public void m( ) { /* body of method*/ }
}
```

The expression A.m(. .) is unable to detect the method invocation in:

B b = new B( );
b.m( );
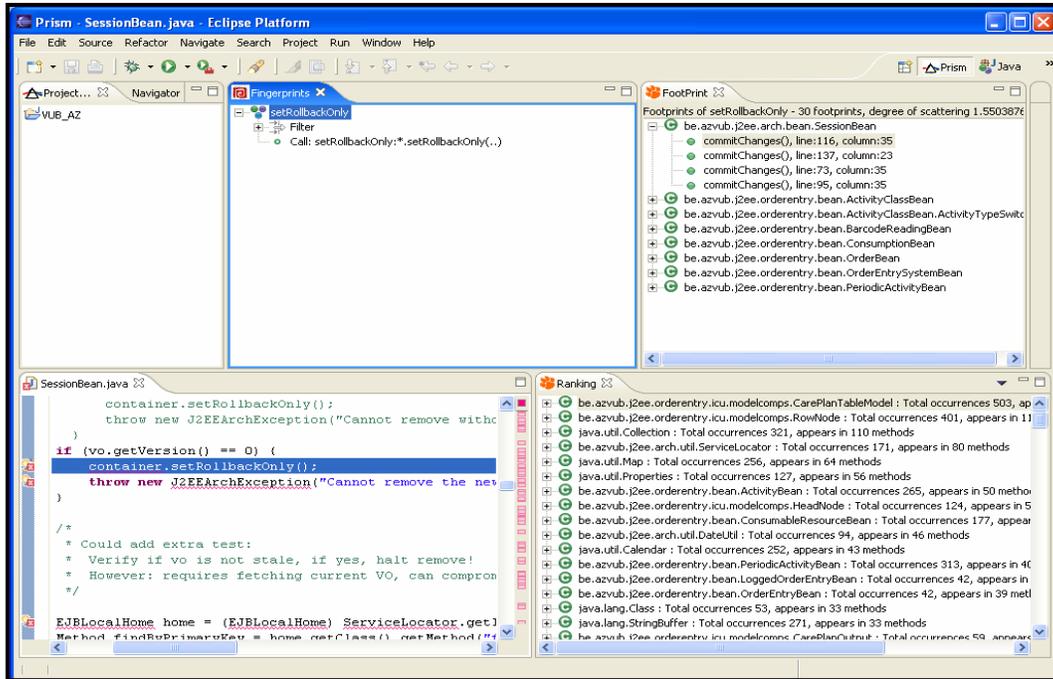
Figure 3.3: Prism Perspective

3.1.1.5 JQuery

JQuery [EV04] is a flexible, query-based source code browser, developed as an Eclipse plugin. A JQuery developer can define his or her own top-level browsers on-the-fly by formulating logic queries and running them against the source code. Alternatively, the developer can select from a variety of pre-written browsers, and use them as-is or modify them to suit specific needs. In this manner, JQuery provides the developer with a wide variety of crosscutting as well as non-crosscutting views within a single tool. Elements in the tree can then be queried individually in the same manner allowing further exploration of the complex web of relationships that exist between scattered elements of code, without the distraction of switching tools or losing the context of the original query. The JQuery query language is a logic (Prolog-like) query language based on TyRuBa. TyRuBa is a logic programming language implemented in Java. The JQuery query language is defined as a set of TyRuBa predicates. Before JQuery can query a code base,

the code must be parsed and put into the TyRuBa database, taking advantage of Eclipse APIs for parsing the java abstract syntax tree. The database only needs to be created once per instance of Eclipse because source code change events are sent directly to the database which updates itself on the fly. The results of a query are displayed in a results tree, part of an Eclipse view. Any of the results nodes can be built upon by performing a sub-query, generating a new sub-tree emanating from that node, see figure 3.4 [EV04].

Table 3.4 lists a sample of the predefined predicates in the query language [JV03]. There are several predicates that exceed the essential elements and relationships that are present in a Java code. The method(?M, tag, ?Tag, ?Value) predicate recovers the value of JavaDoc tags attached to method declarations. The predicates of error( ) give access to the position and severity of compilation errors.

To determine dependencies at the class level there is the refType(?Ref, ?Caller, ?Callee) predicate that determines references to every fields and methods contained in a particular type. The predicates in the query language follow the convention that the names of the predicate correspond to the type of an object and the parameters correspond to, respectively, an object reference, an attribute name or relationship name, and a value. For example, class(?C, name, X) is a query that discovers all classes ?C who's name property is X.

Note that TyRuBa has non-standard lexical conventions for the denotation of variables and constants. In TyRuBa, symbols starting with a "?" are variables. This is convenient because Java identifiers indicating class, field and method names can be used as constants.

JQuery has a numbers of advantages, like:

- Merge the feature of query based tools and hierarchical browser tools.
- The result of the query is used to define a first browser view that serves as a starting point for a discovery process.
- The developer can navigate the tree and extend it at will by requesting extra queries to be added as sub trees of particular nodes of interest.

- Decrease require to exchange between different views. This avoids the confusion caused by exchanging views and keeps an unbroken representation of the whole search path.

JQuery has a numbers of disadvantages, like:

- When the tree is expanded several levels deep, it tends to become too wide and too cluttered to fit in the JQuery pane. To obtain an overview it is needed to scroll the view horizontally and vertically. This is awkward and makes it harder to understand the relation between elements separated by several levels in the tree.
- The developer needs to be customary with Eclipse Platform.
- The logic query language is very difficult to use for complex queries.
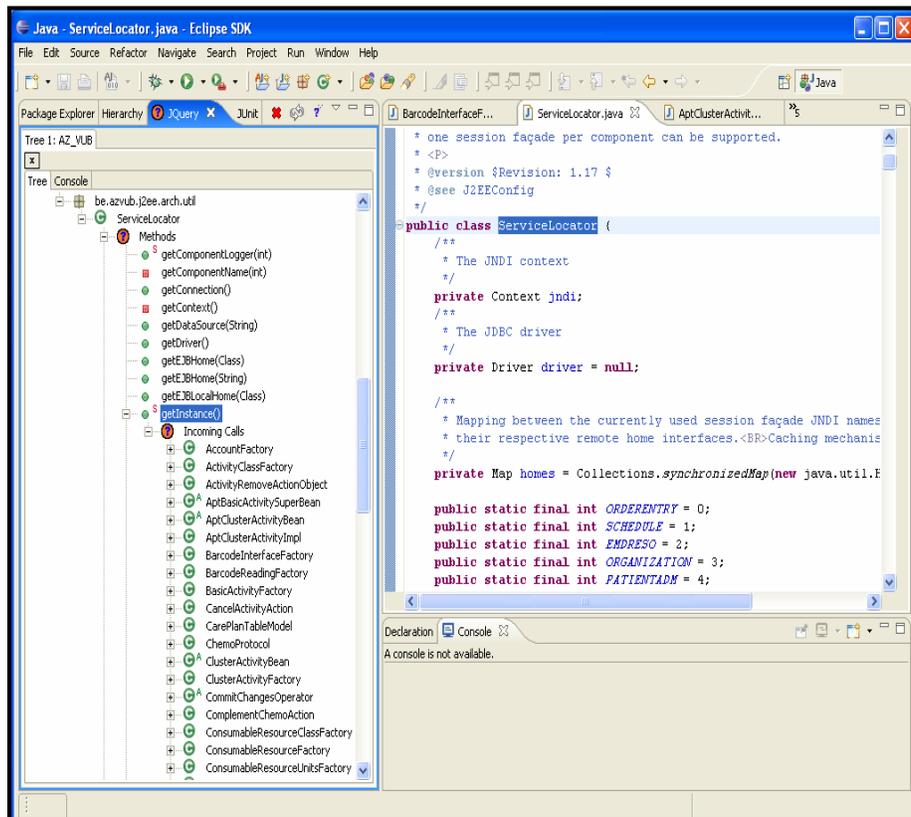
Figure 3.4: JQuery Perspective

Predicate	Description
package(?P)	True if ?P is a package.
package(?P, name, ?N)	True if package ?P has name ?N.
package(?P, type, ?T)	True if package ?P contains type ?T.
type(?T)	True if ?T is a type.
type(?T, name, ?N)	True if type ?T has name ?N.
type(?T, field, ?F)	True if type ?T contains field ?F.
type(?T, method, ?M)	True if type ?T contains method ?M.
type(?T1, type, ?T2)	True if type ?T1 contains inner type ?T2.
type(?T, modifiers, ?M)	True if type ?T has modifiers ?M, where ?M is a list.
type(?T1, super, ?T2)	True if type ?T1 has super type ?T2.
type(?T, tag, ?Tag, ?Val)	True if type ?T has a JavaDoc tag ?Tag with value ?Val
class(?C1, extends, ?C2)	True if ?C1 extends class ?C2.
class(?C, implements, ?I)	True if class ?C implements interface ?I.
class(?C, creator, ?M)	True if an instance of class ?C is created in method ?M.
method(?M, returnType, ?RT)	True if method ?M has return type ?RT.
method(?M, paramType, ?PT)	True if method ?M has a parameter of type ?PT.
method(?M, exception, ?ET)	True if method ?M throws an exception of type ?ET.
method(?M, tag, ?Tag, ?Val)	True if method ?M has a JavaDoc tag ?Tag with value ?Val
refMethod(?R, ?Cler, ?Clee)	True if ?R is a reference from method ?Cler to method ?Clee.
error(?E, message, ?M)	True if error ?E is described by message ?M.
error(?E, severity, ?S)	True if error ?E has severity ?S.

Table 3.4: Some predefined predicates in the query language [JV03]

Comparison of the dedicated browsers

Table 3.5 shows certain of search capabilities for the dedicated browsers discussed above.

	Search Abilities			
	Text-based Analysis	**Type-Based Analysis**	**Method call**	**used wildcards**
FEAT	n/a	n/a	n/a	n/a
Aspect Browser	√	×	×	"*"
AMT	√	√	×	nothing
Prism	×	×	√	"*", ".."
JQuery	n/a	n/a	n/a	n/a

Table 3.5: Comparison of dedicated browsers (part 1)

	Browsing Abilities	Valid characterization constructs	Additional analysis achieved
FEAT	-Java constructs -Relationships	-Java constructs -Relationships	Compare between two concerns
Aspect Browser	n/a	Text-based Analysis	-Match count -Redundant lines of code
AMT	n/a	-Text-based Analysis - Type-Based Analysis	×
Prism	n/a	Method calls	Ranking advisor
JQuery	-Java constructs -Relationships	Logic source-based queries	×

Table 3.5: Comparison of dedicated browsers (part 2)

- **n/a:** not allowed.
- **Java constructs** type, method and field.
- **Relationships:** declare, declared by, calls, called by, etc.

3.1.2 Automated aspect mining techniques:

We can use automated aspect mining techniques to aid developers for automate determine starting points or seeds to mine candidate aspects. We know that dedicated browsers need seeds of a concern to search manually by browser for those candidate aspects in legacy code [KM05]. Consequently, we can use automated aspect mining techniques to aid developers to determine seeds in order to mine candidate aspects. In this kind of approach there are advantages and disadvantages: the advantage is that no input or query is required from the developer in order to identify concerns. However, the disadvantage is that only very common concerns are likely to be found, and code which implements a given concern, but even slightly deviates from the pattern encoded in the tool, is likely to be missed [HT05]. There are several examples of automated aspect mining techniques and tools like:

3.1.2.1 Analyzing recurring patterns of execution traces.

Technique

Breu and Krinke offer a technique based on program traces. A program trace is a series of method calls and exits. In these traces they identify recurring execution patterns which

describe certain behavioral aspects of the software system. They assume that recurring execution patterns are potential crosscutting concerns which describe recurring functionality in the program and thus are possible aspects. In order to search these recurring patterns in the program traces, a classification of possible pattern forms is required. Consequently, we present the idea of execution relations between method calls [Bre04]. Consider the following example of an event trace, where the capitals represent method names [KM05]:

```
B( ) {
    C( ) {
        G( ) { }
        H( ) { }
    }
}
A( ) { }
```

Breu and Krinke distinguish between four different execution relations: outside-before (for example, B is called before A), outside-after (for example, A is called after B), inside-first (for example, G is the first call in C) and inside-last (for example, H is the last call in C) [KM05]. By using these execution relations, their mining algorithm searches aspect candidates based on recurring patterns of method calls. If an execution relation occurs more than once, and recurs uniformly (for instance, every call of method B is followed by a call of method A), it is considered to be an aspect candidate. Of course, to make sure that the aspect candidates are suitably crosscutting, there is an additional requirement that the recurring relations should show in different 'calling contexts' [KM05].

Tool support (DynAMiT)

DynAMiT (**Dyn**amic **A**spect **Mi**ning **T**ool) is considered to be the first aspect mining tool that is able to identify automatically both seeded and existing crosscutting concerns in legacy systems based on dynamic analysis [Bre04].

3.1.2.2 Formal concept analysis

The technique of formal concept analysis (FCA) is rather simple. Starting from a (potentially large) set of objects and attributes of those objects, FCA determines maximal groups of objects and attributes. These maximal groups are called concepts. Each such concept consists of set objects that have one or more attributes in common and such that no other objects have those attributes nor are there any other declared attributes they have in common [TM04].

3.1.2.2.1 Formal concept analysis of execution traces (Dynamic analysis)

Technique [CMM+05]

Dynamic analysis is the observation of the runtime behavior of a software system. The runtime behavior is analyzed by using execution traces. These are obtained by running an instrumented version of the program under analysis, for a set of scenarios (use-cases).The execution traces linked with the use-cases are the objects of the concept analysis context, whereas the executed methods are the attributes. In the resulting concept lattice (with 'sparse labeling'), the use-case specific concepts are those labeled by at least one trace for a certain use-case (for example, the concept contains at least one specific attribute) while the concepts with zero or more attributes as labels are regarded as generic concepts. Thus, use-case specific concepts are a subset of the generic ones. Both use-case specific concepts and generic concepts take information potentially useful for aspect mining, since they group specific methods which are always executed under the same scenarios. When the methods that label one such concept (using the 'sparse labeling') crosscut the principal decomposition, a candidate aspect is determined.

Tool support (Dynamo)

Dynamo is a tool for the identification of aspects in the existing Java code. Traces of Execution are produced for the use cases which exercise the principle functionalities of a given application. The relationship between the traces of execution and executed computational units is subjected to concept analysis. In the resulting lattice, potential aspects are detected by determining the use-case specific concepts and examining their

specific computational units. When these come from multiple classes, which in turn contribute to multiple use-cases, a candidate aspect is recognized [dyn].

3.1.2.2.2 Formal concept analysis of identifiers (Identifier Analysis)

Technique [KM05]

Tourwé and Mens offer an alternative aspect mining technique which is based on formal concept analysis. Their technique performs an identifier analysis by using the FCA algorithm. The assumption behind this technique is that interesting concerns in the source code are reflected by the use of naming conventions in the classes and methods of the system. Like input to the FCA algorithm, the classes and methods in the system are employed as objects. As attributes, the FCA algorithm employs substrings produced from the classes and methods' names. For instance, a class called QuotedCodeConstant is split in the strings 'Quoted', 'Code' and 'Constant'. Substrings with little meaning, like 'a', 'with', . . . are discarded from the results. The resulting concepts consist out of maximal groups of classes and methods which share a maximal number of substrings. After having filtered out many unimportant concepts automatically, a significant number of concepts remain which need to be inspected manually. Apart from being able to detect a number of programming idioms, design patterns and certain refactoring opportunities, the same technique can be used for aspect mining purposes by restricting the concepts to those that are crosscutting (for example, the involved methods and classes belong to at least two different class hierarchies).

Tool support (DelfSTof)

DelfSTof developed by Tourwé and Mens's. It presents the discovered concepts in a way that is easy to use and manipulate. It consists of an efficient FCA algorithm, a set of filters, and a set of 'analyzers' that are in charge of the classification, combination and annotation of concepts. They capitalize the letters "ST" because the tool is implemented completely in Smalltalk and originally only analyzed Smalltalk source code [MT05].

3.1.2.3 Natural language processing on source code

Technique

Developers often use Natural Language Processing (NLP) clues to aid understand software; because NLP aids them identify concepts that are semantically related. Shepherd, Tourwé, and Pollock use a NLP technique called lexical chaining to identify groups of semantically related source code entities, and they evaluate whether those groups represent crosscutting concerns. To find crosscutting concerns we look for chains that have members with a high amount of scattering. They think that these chains will often correspond to high level concerns that are scattered throughout code [STP05].

A chainer takes as input a text and groups every word in that text in a chain with closely related words also appearing in the text. It outputs a list of chains that each contains closely related words. In order to compute lexical chains, we need to be able to calculate the semantic distance, or the strength of relationship, between two given words. Researchers have shown that it is easy for humans to determine semantic distances between two, closely related words, and that they do so with reasonable consistency. However, semantic distance is more difficult to determine computationally. In order to compute the distance automatically, a database of known relationships between words, such as WordNet, is often used. The semantic distance between two words is then approximated by using the lengths of the relationships path between the two words in WordNet [STP05]. In order to mine for crosscutting concerns, Shepherd, Tourwé, and Pollock apply the chaining algorithm to the comments, method names, field names and class names of the system they are analyzing. A developer of their approach needs to manually inspect the resulting chains in order to select likely aspect candidates [KM05].

3.1.2.4 Detecting unique methods

Technique [KM05]

Gybels and Kellens offer the use of heuristics to mine for crosscutting concerns. They observe that, in pre-AOP days, crosscutting concerns were often implemented in an idiomatic way. Certain of these idioms can be considered as "symptoms" of aspect candidates. An example of such an idiom is the implementation of a crosscutting concern by means of a single entity in the system which is called from many places in the code

(for instance, a 'logging' entity which is called from throughout the code) see figure 3.5. To detect instances of this pattern, Gybels and Kellens offer the "Unique Methods" heuristic which is defined like: "a method without a return value which implements a message implemented by no other method".

Tool support (unique method)

Gybels and Kellens applied the unique method technique on an entire Smalltalk image [GK05]. After selecting all the Unique Methods in legacy system, sorting them according to the number of times a method is called, and filtering out irrelevant methods (like for instance accessor and mutator methods), the developer has to manually inspect the resulting methods in order to find suitable aspect candidates [KM05].
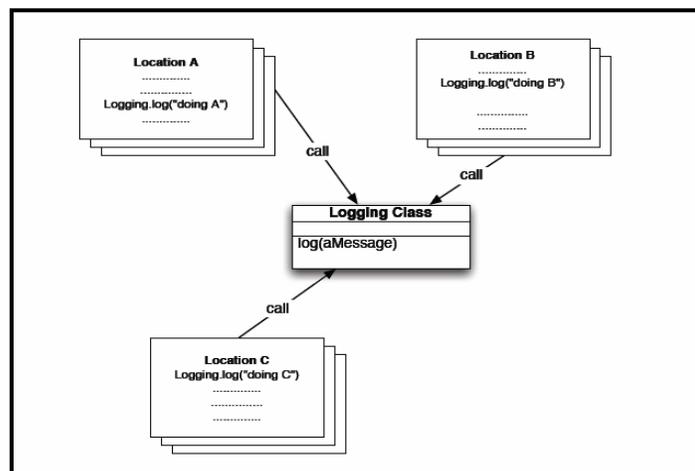
Figure 3.5: Logging as a central class providing logging functionality [GK05]

3.1.2.5 Clustering (Hierarchical clustering of similar method names)

Technique

Shepherd and Pollock perform agglomerative hierarchical clustering (ALC) in order to group methods. ALC first places every object (in this case, every method in a program) that it will cluster in its own group. Then, it repeats the following steps [SP05]:

Step 1. Compare all pairs of groups using a distance function; mark the pair that is the smallest distance apart.

Step 2. If the marked pair's distance is smaller than a threshold value, merge the two groups. Otherwise, stop the algorithm.

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

Tool support (AMAV)

Shepherd and Pollock [KM05] used the above technique as part of an aspect-oriented IDE named AMAV (Aspect Miner and Viewer), which allows for easy adaptation of the distance measure used by the algorithm. For a first experiment they used a simple distance measure opposite proportional to the common substring length of the names of the methods. This mining algorithm is used in combination with the viewing tool of the IDE which not only displays all the clusters which were found, but also consists out of the crosscutting pane and the editor pane. The crosscutting pane displays all methods which are related to a cluster's implementations. This pane, although lacking the context of each method (for example, its class), allows the developer to check the consistency of the concern. The editor pane displays the class context (Java file) for a particular method. It allows the developer to edit a method's implementation with the crosscutting and class context available [SP05].

3.1.2.6 Fan-In analysis
Technique

The fan-in analysis technique is an approach based upon the observation that code implementing a crosscutting concern is often called from different places throughout the software system at hand, thus revealing there is a scattered similar functionality, which is at the same time tangled with the main concerns of the system. This method calling situation is known as fan-in metric, and is defined by Marin et. al as: the number of distinct method bodies that can call a method m. An important consideration with the preceding definition is that 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. We use the code of figure 3.6

as an example to illustrate this situation. 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 table 3.6. Observe that the call in f2 to B's m contributes to the fan-in of m in B's supertypes (A) as well as its subclasses (C1 and C2) [MDM04].

Marin et. al state that high fan-in values indicate the presence of crosscutting concerns in the following situations [MDM04]:

- The high fan-in method is a key element of the aspect implementation, such as the output method for logging, tracing or debugging functionalities.

- The crosscutting implementation is scattered over the system and relies on common functionality and the high fan-in method is part of this functionality.

- 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.

```
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 3.6: Various (polymorphic) method calls [MDM04]

Method	Caller set	Fan-In value
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

Table 3.6: Fan-in values for code in figure 3.6 [MDM04]

Fan-in analysis generates candidates based on the fan-in metric of a method: if a method is called from many, scattered places, the method is considered a potential seed. Consequently fan-in is essentially a metric for the scattering symptom of the crosscutting concerns [Mar].

Marin et al. describe a number of properties which must be considered for analyzing the callers of a candidate. These properties show possible relations between the callers of a method with a high fan-in, and aimed at reducing the percentage of unimportant caller for reasoning. The list of proposed properties comprises [Mar]:

- Structural relations between the callers. These relations include:
  - *Same hierarchy*: The methods (callers) are defined by the same interface (super class). As a particular case, the callers could be implementations of the same method.
  - *Common roles*: A method is associated with all the roles applied by its class, so that the methods can share common roles. A role is typically defined by an interface. The methods which belong to the same hierarchy will also share the role which defines the hierarchy.
  - *The same class*: The callers belong to the same class, as for the case of a class level contract.
- Consistent call position: The position of the call, relative with the body of the caller, is consistent for the callers of the method reported to a high value of fan-in.
- Naming-based relations: The callers have similar names. The naming-based and the structural relations can be expressed by an AspectJ-like the definition of pointcut, whereas the position of call could be an indication of the advice type (before/after).
- Relations based on the structure of the call: similar call-sites. An example is the exception wrapping concern that consists of catching a specific type of exception and re-throwing an exception of a different type.
- Intentional relations between callers, such as modifiers of Subject objects in the context of the Observer model. The relations between the callers are due to their participation in the model implementation.

Tool support (FINT):

Marin et al. developed a tool called FINT (see figure 3.7). It is available as an Eclipse plugin. It automatically calculates 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 see figure 3.8 [MDM04].

FINT view has three panes: "Calee Filters Setup", "Caller's Filters Setup", and "Save Results To". In the both panes "Calee Filters Setup" and "Callers Filters Setup", we can mark particular packages or classes to be skipped elements in the fan-in analysis. In "Save Result To" pane, we can specify the file location where the result of fan-in is saved. In the right side of FINT view there are two items: The first one is check box, labeled with "Accessors", that is used to filter getters and setters methods from fan-in analysis. The second item is an input field used to specify a value as threshold to display all methods that have a fan-in greater than or equal this threshold.

The fan-in identification process. Fan-in analysis performs its mining process in the following steps [MDM04]:

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 greater than or equal a certain threshold.
- 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.
- 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.
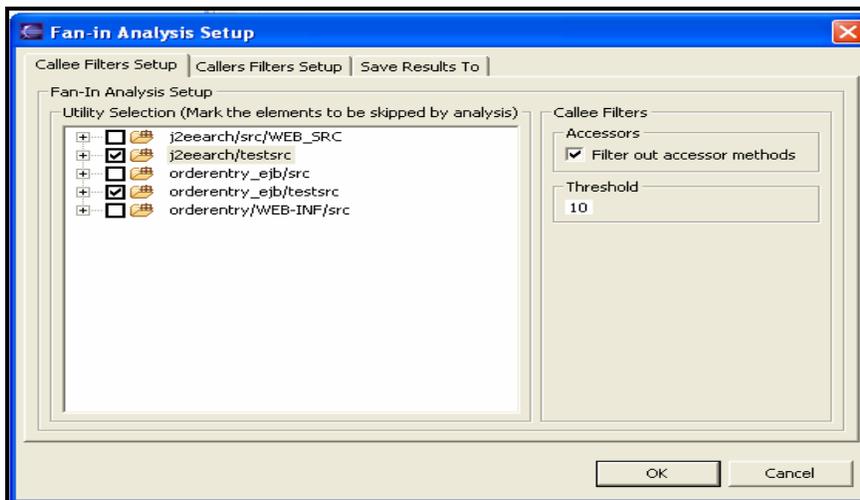
Figure 3.7: FINT view
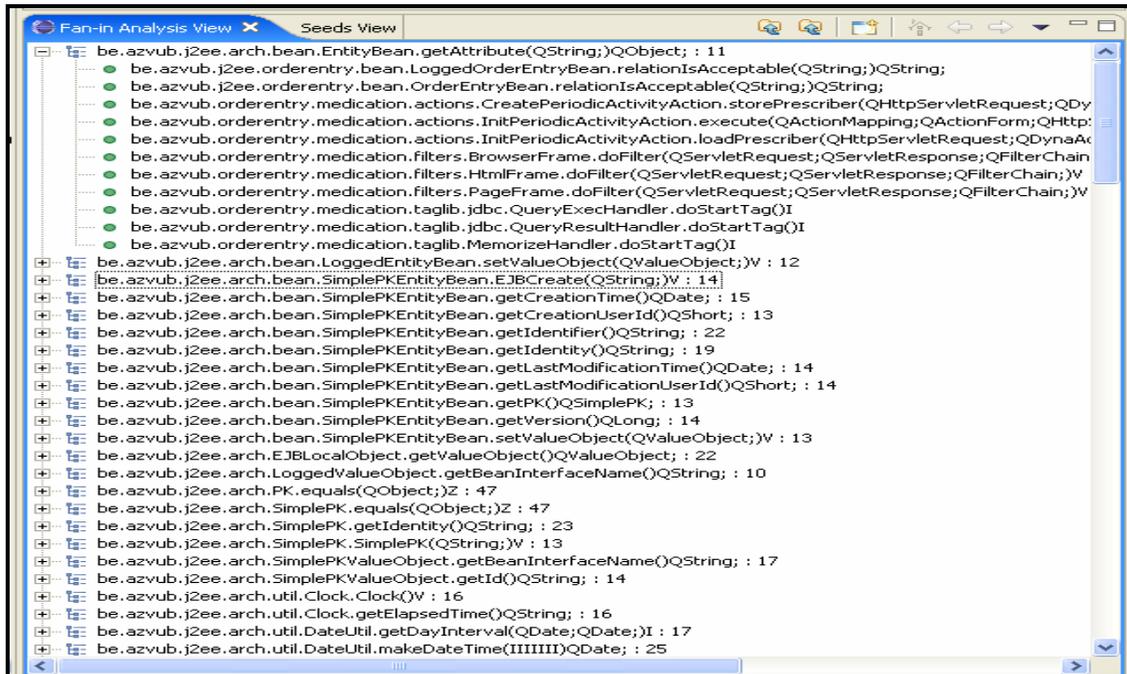
Figure 3.8: Results of FINT

3.1.2.7 Clone detection

Technique:

Clone detection is an active branch in software (re)engineering research that deals with finding parts of duplicated code in systems. A code clone is a code portion in source files that is identical or similar to another. Clones are introduced because of different reasons

such as reusing code by 'copy-and-paste', etc. Clones make the source files very hard to modify consistently. For example, let's assume that a software system has several clone subsystems created by duplication with slight modification. When a fault is found in one subsystem, the developer has to carefully modify all other subsystems. For a large and complex system, there are many developers who take care of each subsystem and then modification becomes very difficult [KKI02]. Clone detection can be used as a technique for aspect mining, since (portions of) cloned code can be considered as seeds or starting points to mine candidate aspects. A seed in the context of aspect mining consists then of "The identification of a method, interface or group of statements that are part of the concern's implementation" [MDM04].

There are several clone detection techniques such as [BDET04]:

- *Text-based techniques* perform little or no transformation to the 'raw' source code before attempting to detect identical or similar (sequences of) lines of code. Typically, white space and comments are ignored.

- *Token-based techniques* apply a lexical analysis (tokenization) to the source code, and subsequently use the tokens as a basis for clone detection.

- *AST-based techniques* use parsers to first obtain a syntactical representation of the source code, typically an abstract syntax tree (AST). The clone detection algorithms then search for similar sub trees in this AST.

- *PDG-based techniques* go one step further in obtaining a source code representation of high abstraction. Program dependence graphs (PDGs) contain information of semantically nature, such as control and data flow of the program.

- *Metrics-based techniques* are related to hashing algorithms. For each fragment of a program the values of a number of metrics is calculated, which are subsequently used to find similar fragments.

Tool support

There are several clone detection tools such as: Duploc[BB02], JPlag[BB02], Moss[BB02], CloneDrm[BB02], and CCFinder[BB02] tools, we will only explain CCFinder, because it able to handle software projects regardless their size, and it detects clones in the language subject of our research, namely Java.

CCFinder [KKI02]:

We start by explaining a number of important concepts used in CCFinder's:

- *Clone relation.* A clone relation is defined as an equivalence relation (for example, reflexive, transitive, and symmetric relation) on code portions. A clone relation holds between two code portions if (and only if) they are the same sequences.

- *Clone pair.* For a given clone relation, a pair of code portions is called clone pair if the clone relation holds between the portions.

- *Clone class.* A clone class is a maximal set of code portions in which a clone relation holds between any pair of code portions. For example, suppose a file has the following 12 tokens: X A B C Y A B C Z A B K: We get the following three clone classes:

    Class1: X <u>A B C</u> Y <u>A B C</u> Z A B K

    Class2: X <u>A B</u> C Y <u>A B</u> C Z <u>A B</u> K

    Class3: X A <u>B C</u> Y A <u>B C</u> Z A B K

Clone detection process see figure 3.9. The entire process of CCFinder's clone detecting technique consists of four steps:

- *Lexical analysis:* Each line of the source files is divided into tokens corresponding to the lexical rules of the programming language and white spaces are removed. This results in a token sequence containing the concatenation of all tokens.

- *Transformation:* The tokens are transformed by transformation rules. These rules are language specific, for example, in Java "name1.name2.name3" will result in "name3", and thus the package is ignored. Then, each identifier related to types, variables and constants is replaced with a special token. As a result code portions with different variables can be recognized as clone pairs.

- *Match detection:* From all the substrings on the transformed token sequence, equivalent pairs are detected as clone pairs.

- *Formatting:* Each location of a clone pair is converted into the line numbers on the original source file.

48

Output results: The output of CCFinder is a text file with the following sections:

- *Option section:* Including the version number of CCFinder, the language specification, etc.

- *Input files section:* Including the paths of the input source files.

- *Errors section:* Including locations at which the lexical analyzer reports some errors.

- *Clones section:* Including the maximal clone pairs.
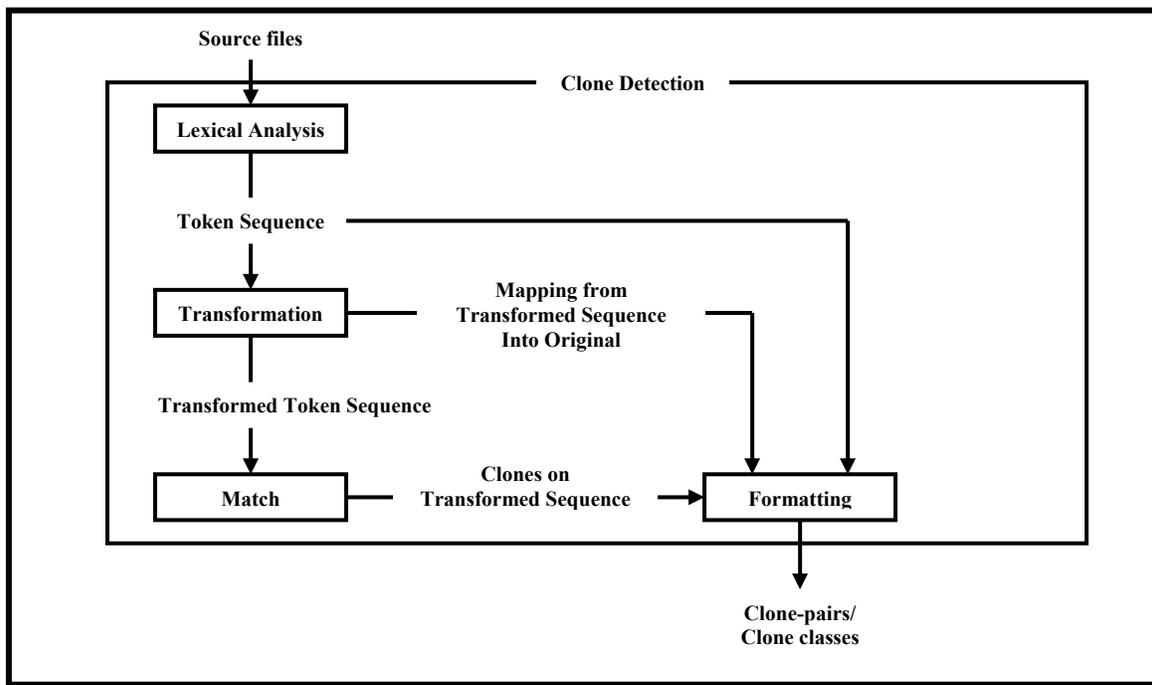
Figure 3.9: Clone detection process [KKI02]

CCFinder results can be visualized through a graphical interface called **Gemini**. Next those clones have to be analyzed manually to find possible aspects.

3.2 Refactoring

3.2.1 Object –Oriented Refactoring

Definition

In order to maintain software its structure often needs to be improved. To automate this, Fowler defines the refactoring process as ["***Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.***"[Fow00]]. Typical examples are operations such as replacing direct uses of public fields with accessors and modifiers, or creating an abstract super class to encapsulate common behavior in similar classes.

The refactoring is an incremental process achieved by performing a series of small steps, each doing a single transformation of the system.

Why to Refactor

Most of the software systems spend long time in a maintenance phase to fix the bugs introduced through out the software system's life time, or to add new features for meeting changing requirements. All of these activities mean modifying or extending existing code. So the readability and maintainability of the code base should be the primary features of any developed system.

Where to Refactor

Refactoring is a process that will help achieve that. Fowler introduces the concept of bad smells describing areas, in the code that suffer from bad design where the refactoring process could be taken. Examples of such bad smells are duplicated code, long method, large class, long parameter list, and etc.

Refactoring Techniques

We will explain same of the OOP refactoring techniques that are presented by Fowler.

- *Extract Method*: Probably the most used technique when refactoring a method which suffers has the "method too long" smell. So we create a new method, and extract a portion of code out the long method, and put it to the new method. Extracting the relevant code out into its own method allows it to be called

somewhere else, and makes the original method easier to read. There is a problem in front of the extracted method in dealing with the local variables of the original method that became out of the scope of extracted method. The solution of that problem is applying other refactorings for instance *Split Temporary Variable* and *Replace Temp With Query*.

- *Extract class*: This refactoring is used when a class is violating the principle of *separation of concerns*. That is, the class is implementing multiple concerns that should be divided into two or more classes.

- *Inline Class:* This is the opposite of Extract Class, and should be applied when a class is doing too little, so this refactoring is used to move all features of the class (fields and methods) into another class and delete the in-lined class.

These refactorings can be automated so there are existing refactoring tools that now offer a variety of such automated refactorings. Such of these tools exist in the Java IDE of Eclipse (JDT).

Pre/Post-condition

To ensure that the refactoring is not applicable and doing something that leads to inconsistent behaviors, the refactoring must specify and implement a set of pre/post-conditions. These pre/post-conditions ensure that the program's behavior will be correct at the end and the complexity of pre-conditions varies a lot depending on the refactoring. For instance, for renaming a class, refactoring needs to check the precondition that the new name will not clash with an existing class. Also, as a post-condition, any existing reference to the old name should be redirected to the new name.

3.2.2 Aspect-Oriented Refactoring

Aspect-oriented refactoring helps in reorganizing code of crosscutting concerns to improve modularization and get the source-code clear of code-tangling and code-scattering. There are three kinds: Aspect-aware Refactorings, Refactoring to aspects (OO → AO), and AO → AO Refactorings.

Aspect-aware Refactorings

When applying an OOP refactoring to a system with aspects, it might the necessary to adapt the pointcut / advice. To achieve this automatically, the refactoring process must take in account the aspect presence in order to preserve the aspect's behavior. For example when a "Rename method" transformation changes a name of a particular method that is captured by a specific pointcut the pointcut may in this case lose the offered information, resulting that the aspect will behave incorrectly. Hanenberg [HOU03] proposes solutions for such problems by suggesting conditions that must be taken into account to ensure the preservation of aspect's behavior when applying refactorings in aspect-oriented system.

- The number of those join points which are addressed by a particular pointcut is not changed after refactoring.
- Those join points which are captured by a particular pointcut have an equal position within the program control flow in contrast to the state before refactoring.
- The join point information offered by each pointcut does not decrease.

Hanenberg [HOU03] introduces aspect-aware versions of OOP refactorings taken from Fowler such as rename method [Fow00], extract method [Fow00] and move method. The key idea behind these Aspect-aware refactorings is extending traditional refactorings with proper steps to correctly update references in AOP constructs.

Refactoring to aspects (OO → AO)

In addition to the aspect-aware refactorings which make it possible to apply OOP refactorings to preserve the behavior of the system in the presence of the aspect constructs, there are different refactoring approaches are proposed by researchers such as Monterio, Ladded, Hannemann and Hanenberg, to improve the OOP code using AOP constructs. Feature extraction approaches focus on extracting the code elements that are participants of the crosscutting concerns into aspects.

Monterio in his approach introduces a catalog of refactorings for Feature Extraction using the AspectJ language, Monterio talks about new AOP Specific Smells concepts

equivalent to the bad OOP smells proposed by Fowler [Fow00] to spot problems in existing code that could be removed by refactorings.

The Double personality smell describes classes that play multiple roles in the implementation. A class is a double personality if it contains code that implements a second concern not related to the primary concern of the class. Secondary if this role is crosscutting then it can be extracted into an aspect using feature extraction refactorings.

The Abstract Classes smell describes the classes that have abstract interface implemented by other subclasses inheriting the abstract classes. The suggested refactorings for that abstract class is to remove the smell by moving implementation code to an aspect and turning abstract classes into interfaces. The benefit behind this refactorings is that it enables separation of implementation code from declarations in abstract classes and the subclasses become free to inherit from some other class and interfaces.

We will show in the next section some of the refactorings approaches proposed in certain of current AOP research. Monterio [MF05] introduced a Catalog of Aspect-Oriented Refactorings included around eleven AOP refactoring mechanisms focusing on transformations from Java implementations to their AOP equivalents in AspectJ. A couple of examples of such mechanisms are:

- *Extract Feature into Aspect [MF05]*

  This refactoring extracts the feature related to crosscutting concerns that is scattered across several methods and classes or tangled with unrelated code. The main purpose of the refactoring is to transfer all members contributing to that feature to an aspect. This is a composite refactoring that uses other refactorings such as *Move Field From Class to Inter-Type Declaration*, *Move Method From Class To Inter-type D*eclaration *and extract advice* [MF05].

- *Move Method From Class To Inter-type Declaration[MF05]*

  Move a method addressing the secondary concern in a class to an aspect by using an inter-type declaration, such that the method can be integrated back with owner class.

- *Extract Advice [HOU, MF, Mon04]*

  This refactoring extracts fragments of code related to a secondary role found in methods of one or more classes. The fragments could be duplicated in a set of methods such as condition statements that appears at the beginning of methods to check the validity of input parameter values. These fragments can be extracted into advice triggered at appropriate join points matching the original locations from which the fragments are extracted.

- *Change Abstract Class to Interface [MF, Mon04]*

  Remove the abstract class by moving implementation code to an aspect and turning the abstract class into an interface. The idea behind this refactoring is that, by separating implementation and declarations in abstract classes, the subclasses become free to inherit from some other class and interfaces.

Ladded [Lad03] in his series of tutorials related to AOP refactoring proposes several AO refactoring techniques such of these refactorings are listed below**:**

- *Extract method calls [Lad03]*

  It is considered a core refactorings used to extract scattered calls of a particular method into advice. For example, calls to a log method for the logging concern can be scattered over the whole working system to log the actions running in the system. These calls to the log method can be extracted into advice. Figure 3.10 shows how we can extract method call from base code into advice. As seen in the figure, the method calls are scattered thought the source code, by extract *method calls* refactoring, these calls can be picked out and added in one location (advice body)

- *Extract exception handling [Lad03]*

  This refactoring is applied to extract exception handling code into a separate aspect.

- *Extract concurrency control [Lad03]*

  Implementing concurrency control requires code to be scattered over many methods. AOP offers reusable implementations for organizing 'acquire' and 'release' implementations of locks: read lock and write lock.

- *Extract contract enforcement [Lad03]*

  Extract tangled code checking pre- or post- conditions for values of the input parameters or return values into a separate aspect.
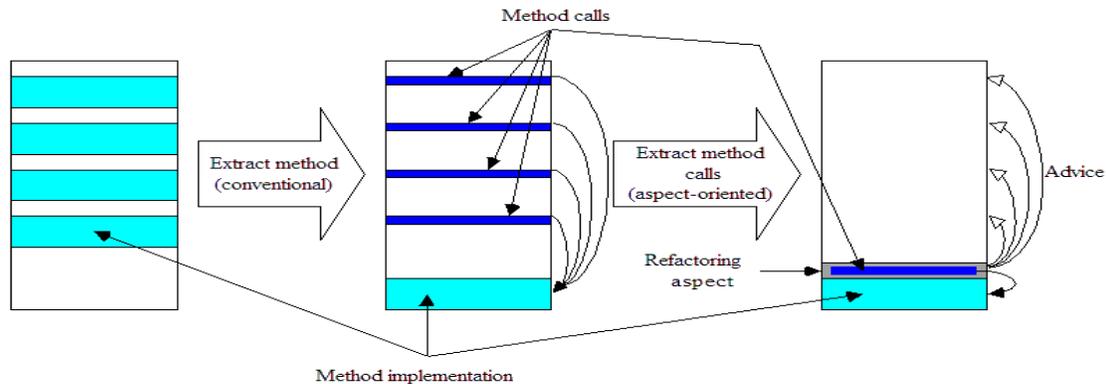
Figure 3.10: Visualization of extract method call into advice [Lad03]

Binkley *et al*. [BCH+05] have developed a partially implemented, human-guided (semi-automated) approach to support OO-to-AO refactorings. Their approach focuses on code bases in which OOP code-blocks related to the implementation of crosscutting concern are identified (marked). Binkley's refactoring is restricted to the replacement of the marked OOP code fragments with one of the following AOP refactorings including pointcut and advice pairs; a refactoring can be useful when the following applicability conditions coupled with the marked fragment of code be appropriate.

- *Extract Beginning and Extract End [BCH+05]*

  If the marked fragment is at the beginning or end of the body of the enclosing method.

- *Extract Before Call and Extract After Call [BCH+05]*

  The fragment of code is always before or after another call.

- *Extract Conditional [BCH+05]*

  A conditional statement controls the execution of the fragment of code.

- *Pre Return [BCH+05]*

  The fragment   of code is just before the return statement.

- *ExtractWrapper [BCH+05]*

  The fragment of code is part of a wrapper pattern, in which the wrapper code is extracted to aspect.

- *Role-Based Refactoring [Han06,Han05]*

  Hannemann introduced a technique named "Role-Based Refactorings" for extracting the crosscutting concerns based on an abstract model describing crosscutting concern elements called role elements and their relationships. The role elements can be classes, methods, or fields that are sets of program elements building the concern. The key idea of Hannemann's refactorings is describing the concern structure by using the abstract model of role elements included in the concern, and mapping a particular role element for each concrete program elements. Hannemann applies the refactorings in terms of those role elements, in other words the refactoring instructions are defined on each role. An example is given by Hannemann explaining how "Role-Based refactorings" were applied to the logging concern in a banking system. An a first step the concern is described abstractly as consisting of two role methods (for example, getLock(..), releaseLock(..)) and their enclosing type CurrencyControl as a role type .

The next of step which specifying the concrete program elements that play these roles and providing refactoring instructions defined on each role to the concrete element. For example the concrete method acquireLock(Account) plays the role of the role method getLock(..), the refactoring instructions defined on getLock(..) can be applied to acquireLock(Account).

Hannemann uses this refactoring approach for replacing crosscutting OO design pattern implementations with their equivalent AOP implementations.

AO → AO Refactorings

A third kind of AOP refactoring research focuses on refactorings related to aspect themselves in order to improve the structure of the aspects. In this section we explain certain of these refactorings found in the research [HOU, Mon04, Han06]. Monterio proposes in his refactoring catalog five new AOP refactorings focusing on AOP

constructs to improve the internal structure of the aspect by removing bad smells such as duplication in the code yielding from extracting the feature to aspect. He also introduces a new smell term called Aspect Laziness smell describing the aspects that do not hold the full burden of their tasks and instead pass the load to classes, in the form of inter-type declarations adding state or behavior to target classes. Monterio describes the situations in which the Aspect Laziness smell is detected by the following conditions

- The additional state and/or behavior are needed by only a subset of the instances of the target classes.

- The additional state and/or behavior are needed only during certain specific phases in the execution of the program.

- Instances of the target classes (may) require multiple instances of that state and behavior simultaneously.

The main reason of this problem is the static nature of the inter-type declaration and its disability in coping with the dynamic requirement of the target classes. Monterio proposes new AOP refactorings such as "Replace Inter-type Field with Aspect Map" [Mon04] and "Replace Inter-type Method with Aspect Method" [Mon04] as a solution for replacing the existing design with a "mapping logic"[Mon04] that supports the same functionality more flexible and dynamic.

Hanenberg et al. [HOU03] propose one AO→AO refactoring named *"separate pointcut"* that is useful in a situation where parts of a pointcut definition are shared in a set of pointcuts. In this situation we can extract the common part from the pointcuts and put the extracted part in a new pointcut, then reuse this pointcut by combining it back to other pointcuts using the logical operators ||, &&, and ! .

3.3 Java 2 platform, Enterprise Edition (J2EE)

In this section, we give a brief overview of Enterprise Java Beans (EJB), which are used as the underlying technology of the case study we used in our experiment. We also illustrate some of J2EE design patterns, like Service Locator, Value Object, Business Delegate and Session Facade.

3.3.1 Enterprise Java Beans (EJB)

Definition

The Enterprise Java Beans is a Java 2 platform, Enterprise Edition (J2EE) technology. EJB is a server-side component technology, which enable the easy development and deployment of component-based business applications. Applications written using the EJB architecture are scalable, transactional, multi-tier, distributed, portable, secure, and reliable. So the main benefit of EJB is the separation of business logic from system code [RP06].

A typical EJB Architecture is shown in figure 3.11 consists of [Raj]:

- EJB Servers.
- EJB Containers.
- The Home Interface and Home Object.
- The Remote Interface and EJBObject.
- EJBs (Session Beans and Entity Beans).
- EJB clients.
- Auxiliary systems like: the Java Naming and Directory Interface (JNDI) and the Java Transaction Service (JTS).

- *EJB Servers:* These provide services such as raw execution environment, multiprocessing, load-balancing, access of device, provides naming and transaction services and makes containers visible.
- *EJB Containers:* These act as the interface between an Enterprise JavaBeans and the external world. An EJB client never accesses a bean directly. Any bean access is made by container-generated methods which in turn call the methods of bean.

58

The two types of containers are **session containers** which can contain transient, non-persistent EJBs whose state is not saved at all and **entity containers** that contain persistent EJBs whose state is saved between calls.
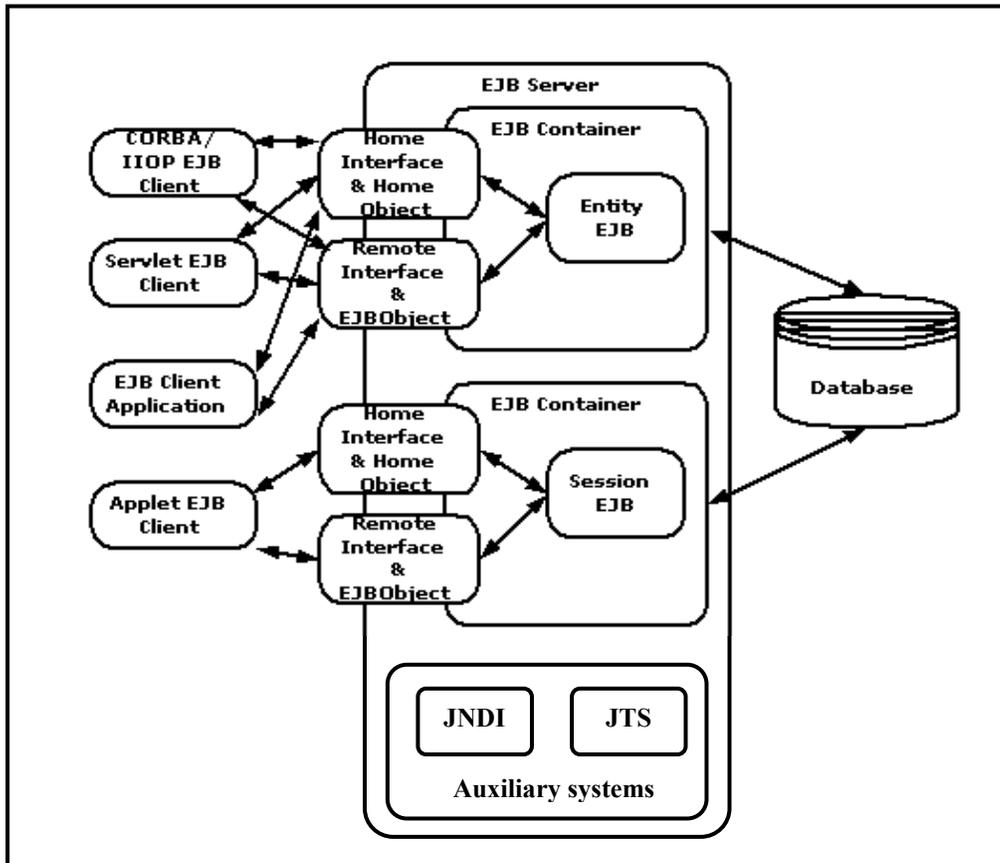
Figure 3.11: EJB Architecture [Raj]

- *The Home Interface and Home Object:* Factory methods to locate, create, and remove instances of EJB classes which are defined in the home interface. The home object is the implementation of the home interface. The EJB developer first has to define the home interface for his bean. The EJB container vendor provides tools that automatically produce the implementation code for the home interface defined by the EJB developer.

- *The Remote Interface and EJBObject:* The remote interface lists the business methods available for the enterprise bean. The EJBObject is the client's view of the enterprise bean and implements the remote interface. While the enterprise

bean developer defines the remote interface, the container vendor provides the tools necessary to produce the implementation code for the matching EJBObject. Note, however, the EJB container is still responsible for managing the EJBObject. Each time the client calls the EJBObject's methods, the EJB container first handles the demand before delegating it to the Enterprise Bean.

- *EJB Clients:* These make use of the EJB Beans for their operations. They find the EJB container which contains the bean by the Java Naming and Directory (JNDI) interface. They then make use of the EJB Container to call EJB Bean methods.

There are two types of EJBs:

- *Session Beans:* Each Session Bean is usually associated with one EJB Client. Each Session Bean is created and destroyed by the particular EJB Client which it is associated with. A Session Bean can either have states or they can be stateless.
- *Entity Beans:* Entity Beans always have states. Each Entity Bean can however be shared by multiple EJB Clients. Their states can be persisted and stored through multiple calls.
  EJB servers have a right to control their working set. *Passivation* is the process by which the state of a Bean is saved to persistent storage and then is permuted outside. *Activation* is the process by which the state of a Bean is restored by permuted it in from persistent storage. Passivation and Activation apply to both Session and Entity Beans.

There are two types of Session Beans:

- *Stateless Session Beans:* These types of EJBs do not have any internal state. Since they do not have any states, they do not need to be passivated. Because of the fact that they are stateless, they can be shared in to service multiple clients.
- *Stateful Session Beans:* These types of EJBs possess internal states. Consequently they must handle Activation and Passivation. However, there can be only one Stateful Session Bean per EJB Client. Since they can be persisted, they are also called Persistent Session Beans. These types of EJBs can be saved and restored

through client sessions. To save, a call to the bean's getHandle() method returns an object of handle. To restore, call the handle object's getEJBObject() method.

Persistence in Entity Beans is of two types:

- *Container-managed persistence:* Here, the EJB container is responsible to save the Bean's state. Since it is container-controlled, the implementation is independent of the data source. The container-controlled fields must be indicated in the Deployment Descriptor and the persistence is automatically handled by the container.

  Note: *Deployment Descriptors* are instances arranged in series of a class. They are employed to pass information about an EJBs preferences and deployment needs to its container. The EJB developer is responsible to create a deployment descriptor along with his/her bean.

- *Bean-managed persistence:* Here, the Entity Bean is directly responsible to save its own state. The container does not need to produce any database calls. Consequently the implementation is less adaptable than the preceding one as the persistence needs to be hard-coded into the bean.

- *Other Auxiliary systems like:*
-   The Java Naming and Directory Interface (JNDI) which makes it possible to Clients of EJB to find beans of EJB.
-   The Java Transaction Service (JTS) that provides the support of transaction in an environment of EJB.

3.3.2 J2EE Design Patterns

In this section, we also give a brief overview for some of J2EE design patterns [pat], which are used in the case study we used in our experiment.

Service Locator

Problem

Enterprise applications need an approach to look up the service objects that give access to distributed components. J2EE applications use Java Naming and Directory Interface

(JNDI) to look up enterprise bean home interfaces, Java Message Service (JMS) components, data sources, connections, and connection factories. Iterant lookup for code makes code hard to read and maintain. Moreover, needless JNDI initial context creation and service object lookups can reason performance problems.

Solution

The Service Locator pattern centralizes distributed service object lookups, provides a single point of control for service access, and may act as a cache that removes redundant lookups. It also encapsulates complexity of lookup and creation process.

Value Object

Problem

Application clients need to exchange data with EJBs. Using several calls to obtain methods that return single attribute values is inefficient and sucks up network bandwidth.

Solution

Create a Value Object (a serializable class with public attributes) that can be used to house all the attribute values of an EJB. The client makes a single remote method invocation. The EJB initializes an instance of the Value Object and passes it by value to the client so this mechanism facilitates data exchange between tiers.

Business Delegate

Problem

Presentation tier components interact directly with business services through RMI. This produces undue coupling, client complexity (networking issues), and poor performance (too many remote calls). The client is tightly coupled to the EJB layer, creating dependencies between client and server that affect both development, run-time and project management concerns.

Solution

Create a Business Delegate to hide underlying implementation details (such as lookup and access of EJBs). The Business Delegate is a client-side abstraction for the server-side services. It hides all distribution details, intercepts remote exceptions, performs any retry or recovery operations, throws application level exceptions as needed, and may cache results locally.

Session Facade

Problem

Clients are coupled directly to session and entity EJBs. Tight coupling leads to decrease in flexibility and software design clarity. Fine-grained method invocations overflow the network.

Solution

Create a session bean as a facade to encapsulate the complexity of interactions amongst the server-side business objects participating in a workflow. The Session Facade pattern defines a higher-level business component that contains and centralizes complex interactions between lower-level business components The Session Facade: provides a simpler interface, creates a higher level "business service" abstraction, eliminates the lower level "chattiness" between the client and the server, and clearly centralizes security, transaction control, and relationship management.

Chapter 4

Aspect Mining in AZ-VUB Case Study

This chapter describes our experiences applying aspect mining techniques on an industrial legacy application written in Java. We also discuss the aspect mining tools used in this experiment and the crosscutting concerns identified in the application. At the end of the chapter we give an evaluation of the mining activity.

4.1 Case study system: AZ-VUB application

AZ-VUB is the academic hospital of the Vrije Universiteit Brussel and one of the larger hospitals in Belgium. Like all medical organizations, AZ-VUB is using a computerized system managing the services it provides. The system supports the storage and evaluation of medical data, and helps to support patient care, and resource scheduling supporting the hospital's management activities including appointments scheduling, drug consumption, bed availability and human resources.

In this section we give a brief description of the AZ-VUB application. The AZ-VUB application is written in Java using the J2EE platform [jav]. It is a web-based system. The user-interface parts are constructed using Java Server Pages (JSP) providing the users with an interface through a web browser. One of the main functions of the system is managing the prescription of medicines which need to be provided to patients who are staying at the hospital. The application allows a physician to select from a catalog of medicines which are needed to be prescribed so different JSP pages are accessed depending on which medicine is selected. The business logic of these activities is implemented through the EJBs containers.

We get a part of AZ-VUB application as case study. This part of the application comprises 37 packages including 408 types. These types contain around 4535 methods. The total number of lines of code in these methods is approximately 7622 LOCm. Figure 4.1 shows distributions of lines of code in methods. We observe that there are a lot of methods containing a large number of lines yielding more complexity for the application.

The packages can be sorted by architecture: there are packages containing types defining the structure of J2EE components such as the value objects, session beans

and entity beans. Other packages include types working as utility classes providing several services needed by the application components. There are also packages of types containing the actions triggering the appropriate EJB functionality.
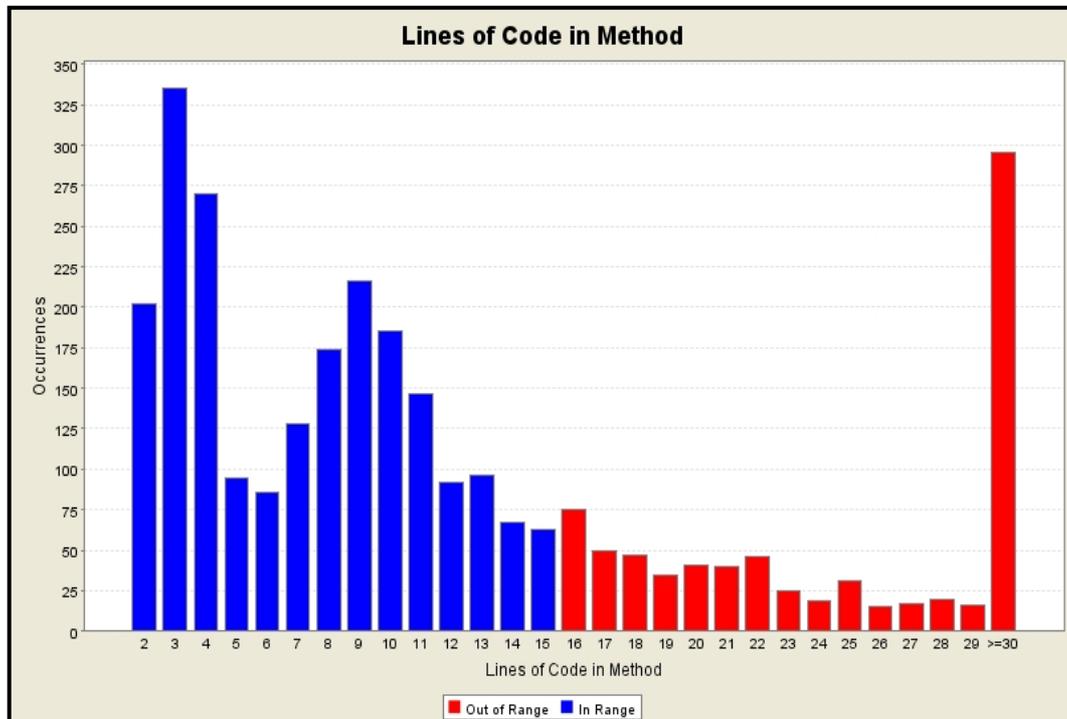
Figure 4.1: Distributions of code lines in methods of the AZ-VUB application

In the next sections we discuss the aspect mining techniques we applied to the case study and discuss the crosscutting concerns we discovered.

4.2 Aspect Mining Approaches

Aspect mining is the discovery of aspects in existing code bases using various set of tools that are discussed in detail in chapter 3(section 3.1). There are two kind approaches of those used tools, approaches called *Bottom-up* approaches discover the concerns automatically and give results advising the developer about the spots of aspect candidates (seeds).

The other kind of approaches is called *explorative* or *query-based (Top-down),* approaches which allow the developer to explore the code or make query on the code bases. In these approaches the developer uses previously identified seeds, or well-known concerns to build a complete outlining the elements and their relationships that are pertinent to the crosscutting concerns.

4.3 Applying Aspects to AZ-VUB application

The idea of applying aspects to AZ-VUB application appears clearly when we consider software evolution. Using AOSD techniques we can adapt the software structure for coping with new variable requirements arising through its lifecycle. We need to modify the software parts that are affected by the changes, modularizing the code related to the same concern and making it more readable and easy to maintain. Therefore, our goals are in the first place, identify which crosscutting concerns can be extracted from AZ-VUB application, then extract these concerns code and apply AOP refactorings to the identified crosscutting concerns, using AspectJ [aspe]. It is important that extracting the aspects does not affect the functionality.

Transforming from OO to AO can be divided into two phases. The first phase is called aspect mining. In this chapter we explain broadly the aspect mining phase illustrating the achieved mining steps and what are used of the mining tools. We analyze the results yielding from applying the tools on the code of the AZ-VUB case.

4.3.1 Used Techniques

In our experiment we used the both kinds of mining approaches: automated tools such as Fan-In tool (FINT) and Prism tool; also we used exploring tools such as FEAT and JQuery. The way that followed by us in the aspect mining of this experiment is using the automated tools at first because these tools takes little input and don't need to be having much knowledge about the application domain or target source code. This way allows us to examine the hot spots that might indicate aspect candidates. At the next step we used the explorative tools to indicate certain elements and their relations of aspect candidate that maybe identified in the first step.

Therefore using automated mining tool would support us with staring points at which we will start our aspect mining and using the explorative tools would allow us to build complete model for the identified concerns illustrating concern relations with other elements. Such as of these relations are same class hierarchies of the method-caller locations (i.e. the methods (callers) are defined by the same interface or super class) for the discovered methods whose high scattering degree in order to help us in extracting these methods. So we discuss in the following sections the applied process of using aspect mining approaches.

All of the used tools integrate tightly into IDE which we use, namely Eclipse. Eclipse [ecl] is an integrated Java development environment. Besides being a good platform for all these other tools, Eclipse is a good development framework in its own right.

4.3.2 Applying Bottom-up Approaches

In this section we present and discuss the mining process using two automated tools on the AZ-VUB case study.

4.3.2.1 Applying Fan-In Tool

We started our mining activity by applying FINT on the case. FINT analyzes the code identifying each method and its number of distinct calls also called fan-in value. Applying FINT on the target case involved the following two steps: [MDM04]

- The first step: we specify the callee sites (packages and classes) in which the called methods that we need to compute the fan-in number; we also specify the caller site in which the methods that call the others in the callee site.

- The second step: we specify a threshold used to filter the results according to fan-in value by showing only the called methods whose high fan-in value above the threshold. Also we can restrict the results by specifying an option for excluding the getter and setter methods.

FINT yields results arranged as a tree structure containing each called method with its fan-in, and it's calling method. The results of FINT needs some effort to manually analyze for examining often called methods (i.e. having high fan-in), which are possible seeds of crosscutting concerns.

Fan-In Analysis

When we applied FINT on the AZ-VUB case, we chose to select methods which have fan-in value above threshold equaling 4. We get 255 methods having fan-in value arranged between 4 and 167. The getter and setter methods are automatically excluded from the results. Figure 4.2 shows a chart illustrating the distribution of these methods (accessor methods are not included) and their fan-in. Form the chart we observe that there are around 138 methods having high fan-in value($>=7$) and representing approximately 3% of the total methods. Table 4.1 shows a part of the FINT results, and their fan-in.
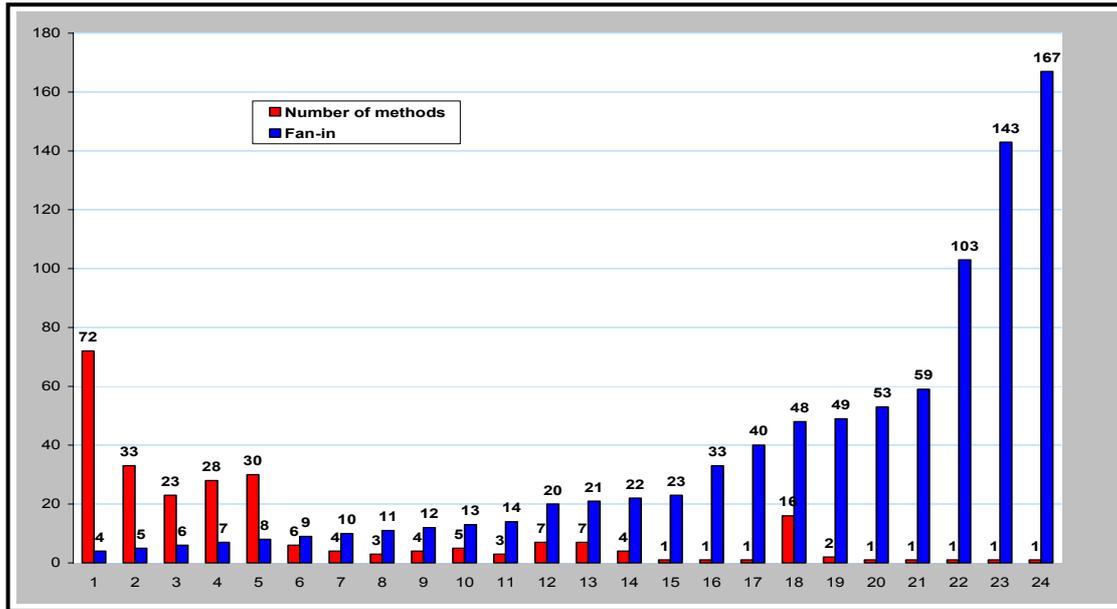
Figure 4.2: Distribution for fan-in on the methods of AZ-VUB application

In our analysis we restricted ourselves to methods returning nothing (void type) because some of these methods are holding actions that are not part of the base code functionality. For methods with no return value it easy to extract them into aspect using a method-to-advice refactoring [GK05].

We also focus on the methods having high fan-in (>= 7). So in our analysis, we examine each called method and it's calling sites to explain if its concern crosscuts the core concern of its calling sites. We observe number of issues through our searching for the references of these methods in the case implementation:

- Because of the polymorphism mechanism, there are some methods could be reported as having high fan-in for other methods, so more carefully analysis are needed to fix this issue.

- There are methods crosscut others but these methods could not be extracted because the difficulties in capturing the necessary context to make robust pointcuts or difficulties in using local variables of the crosscut methods. The local variables are used as variables storing the return values of the crosscutting methods, or as arguments passed to them (the crosscutting methods).

This issue could cause a problem for extracting the crosscutting concern into an advice within aspect, since AspectJ's join point model does not support the referencing for the local variables.

-By examining the call sites of the methods with high fan-in that are implementing checking behavior, we observed that there are some of these methods used in uncommon complex conditions. So this issue raises difficulties in extracting these methods by aspect means.

Next we present some explanation of the identified candidate aspects in order to discuss the effects of the refactoring on the maintainability and evolvability of the application.

Service Locater Pattern

We observe that the methods that implement a J2EE design pattern named Service Locater have high fan-in value. The Service Locater pattern is implemented as a utility singleton class used to manage the session facades for the different components of the AZ-VUB application for caching the resources so as to gain performance. For example the getInstance method in ServiceLocater class has a high fan-in around 184.

Exception Wrapping Pattern

Exception wrapping is a crosscutting concern that affects a number of classes in the system. These classes for instance catch Exception as general exception type thrown by the basic implementation of their methods and then re-throw J2EEArchException as an application specific exception presenting meaningful error messages to users.

This type of the exception manipulating requires a try/catch block in each method through which the wrapping process is established. This is a typical instance of the Business Delegate J2EE pattern [blu] therefore we observe that the constructors of J2EEArchException have high fan-in value.

Notifying Listener Concern

The notifying concern is identified in one class namely CarePlanTableModel. CarePlanTableModel contains methods manipulating the structure of a care plan table storing data shown in Icu care plan frame panes. When the structure of the table changes, a method named fireTableStructureChanged must be invoked to notify table's listener that the data in the table is updated. This method crosscuts eleven methods of the CarePlanTableModel class in different places, for instance after inserting a new row, delete a row, or change the visibility of the table's rows.

Persistence Concern

Class ResultSetWrapper is a wrapper class for java.sql.ResultSet. The class ResultSetWrapper defines getter methods to get result data from ResultSet object that provides access to a table of data generated by executing a query statement. The table rows are retrieved in sequence. Within a row its column values can be accessed in any order. The getter methods retrieve column values for the current row either using the index of the column, or by using the name of the column. In these getter methods we observe that after the column value returned, it is stored in a hash map using either column name or column index as key. These stored values are used to optimize HTML table creation associated with a query's result set.

Other instance of persistence concern is detected in the CarePlaneTableModel class. The CarePlaneTableModel class uses an instance of the "be.azvub.j2ee.orderentry.icu.- IcuRowPropertyProvider" class to load and save properties for a patient. Each patient has a properties file for each Care Unit where he was cared. The IcuRowPropertyProvider.save() method has high fan-in value of 13 and all its calling sits are in the CarePlaneTableModel class. This method uses the care table model to create new properties and then writes them to disk. This method is intended to be called in the table model each time something with the table rows has changed (added, deleted, moved, order changed, visibility changed, etc..).

Consistent behavior concern

There are a number of methods implementing consistent behavior in the execution of other methods. For example, in the CarePlanTableModel class, the recalculateCounts() method with a fan-in value of 8 is called to recalculate both the column count and the row counts of the care plane table each time the status of rows of the table has changed. By examining the call sites, we observed that 4 of them occur at the last of the methods changing the visibility for rows of the table model so it can be easily factored out as an aspect by means of "after" advice.

Contract enforcement

Contract enforcement is crosscutting concern can be observed in several methods in the case study. So there are a lot of methods implementing checking behavior to enforce controlled steps in the executions of others methods for fixing any unexpected results. For example, in the isEmptyString() method with a fan-in value of 9 is used in

the getter methods of the DynaBeanPropertyParser class to check out the null values to prevent unexpected exceptions. By examining the call sites, we observed that all of the method calling occur before calling of the ValueOf(String) method of the wrapping classes of the primitive types (Boolean, Double, Integer, Long, Short, Float). Since the checking condition is similar, it can be easily factored out as an aspect by means of "around" advice for a pointcut capturing the calls of the valueOf(String) methods.

4.3.2.2 Applying Prism Tool

Aspect mining using Prism tool is centered on three concepts: Prism fingerprint, Prism footprint, and Prism advisor. The Prism fingerprint is defined as a regular expression describing abstractly certain elements of a crosscutting concern in the base code. Through the Prism fingerprint, we make queries over the base code searching for crosscutting concern. The result yielding from applying the fingerprint are concrete locations representing either a specific line or a region in the target's source code. We also can use Prism advisors that represent important insights of the structure of the system and, therefore, are supportive in boosting mining procedures and improving the exactness of fingerprint definitions.

We applied Prism tool on AZ-VUB application getting an advisor containing around 210 methods with their total occurrences in the source code. Prism advisors help us by showing a large list of the application elements (types or methods) ranked according to their scattering degree and explain their total of occurrences and where appear in the application code. The total occurrences of the methods look like as the fan-in value yielded by FINT.

We have discovered several crosscutting concerns having elements related to Java-predefined types that did not discovered by FINT tool.

We use the ranking capability of Prism advisor to make prudent guesses. Prism is able to rank the scattering of all class types and their methods used in the system. Types that are used relatively spread throughout the code provide good hints of potential aspects.

Next we will present some explanation for some the candidate aspects identified using the Prism tool.

Transaction Control Concern

We observe that setRollbackOnly() method has high scattering degree and most of its calling sites in OrderEntrySystemBean class. OrderEntrySystemBean class is representing the main facade session for the order entry operations in AZ-VUB application.

The most of the session's methods are tangled with transaction control concern. When the methods performing a transactional process fail through its execution, the rollback action must be taken to undo the uncommitted transaction. We observe that "setRollbackOnly" is invoked on EJB Container object in two catch block trapping the exception thrown. The setRollbackOnly action enforces the EJB Container to rollback the transaction when the failed method exits.

Exception Handling Concern

Exception handling is a crosscutting concern that affects a number of methods in AZ-VUB application. "ActionMapping.findForward(String)" method has total occurrences of 21 and appears in 11 methods. By examining these occurrences sits, we observed that all of them are in "execute" methods of Action classes (11 classes) and used as part of exception handling code.

Also we observed that the operations taken when catching the exception are the same in all the execute methods in the classes implementing the Action interface.

Every execute method in these classes catches the exceptions thrown by the underlying implementation. The problem is that the developer has to write this chunk of code into every execute method. This is not very elegant, and can be easily handled with AOP.

4.3.2.3 Discussion

Table 4.1 summarizes part of the discovered concerns by the both tool FINT and Prism. The first column display names of the concerns. The other columns show by what tools the concern was discovered: if a tool discovered the concern, we put a + sign in the corresponding column, otherwise a - sign is in the table.

We will try to interpret some concerns of what are shown in the table to noticeably identify the strengths and weaknesses of each individual tool.

Concern	FINT	Prism
Logging	+	+
Contract enforcement	+	+
Consistent behavior	+	+
Notifying listener	+	-
Exception wrapping pattern	+	-
Persistence	+	-
Transaction control	-	+
Exception handling	-	+
Service Locater	+	+

Table 4.1: Concerns discovered by both techniques (FINT and Prism)

Logging concern is one instance of the concerns discovered by the both tools. The logging concern comprises methods used to log the occurred events in the system. These methods have high scattering degree so the both tools were able to discover the concern.

Persistence concern comprises two methods (storeField(..)) which are used to store values resulting from querying the database. These methods are tangling the getter methods for only one class (ResultSetWrapper) yielding low scattering degree for the concern, so that concern is discovered only by FINT and not appeared in Prism's result.

Notifying Listener concern is discovered by FINT because the concern method (fireTableStructureChanged) is tangling methods of one class (CarePlanTableModel) so it is not detected by Prism.

Exception Handling concern is discovered in "execute" methods of 11 action classes. These methods use similar way to handle the errors using methods of pre-defined types that were not identified by FINT. Such of these types is "org.apache.struts-.action.ActionErrors" that belongs to struts framework.

Transaction Control is concern discovered by Prism since the concern crosscuts the several transactional methods by invoking the method "setRollbackOnly" on the EJB Container object.

Form the above interpretation for the discovered concern we observe that FINT tool (version 0.3) can only compute fan-in metric for the methods belonging to the selected types in source code of the targeted project. So the source code of the used pre-defined type must be available to be detected by FINT. The aspect seeds of the concerns that encapsulate functionality of the pre-defined types (their source code is not available) are shown by Prism advisor and not appeared in FINT result.

Other observation is that Prism advisor only shows the crosscutting methods that are scattered in multiple classes (at least two classes) so Prism misses some aspect candidates with low scattering degree, such as the notifying concern that crosscuts 12 methods of the "CarePlanTableModel" class. Prism also misses the aspect candidates related to the object creation (class constructors), for example the creation of the "J2EEArchException" exception that is used as part of the exception wrapping concern.

4.3.3 Applying Top-down Approaches

In this section we present and discuss the mining process using two top-down approaches.

4.3.3.1 Applying JQuery Tool

We used JQuery that combines logic programming and Eclipse to produce a way to build interesting views of our source code. For example, we can build queries that view only those classes that implement interface IFoo, etc. Using JQuery we can make dynamic browsers showing the software structure according to particular top-level query. For example we can make the browser present all the methods defined in the system.

Next we give present some concrete examples that illustrate how we used JQuery to explore the case source code.

Example 1:

The example shown in figure 4.3 is one of our searches that query the code structure to find out how Value Object types are constructed. Figure 4.3 is an example of using

a directed query to find a specific opening point in the code from which to explore. So the typed query describes all subtypes of the type named "ValueObject".
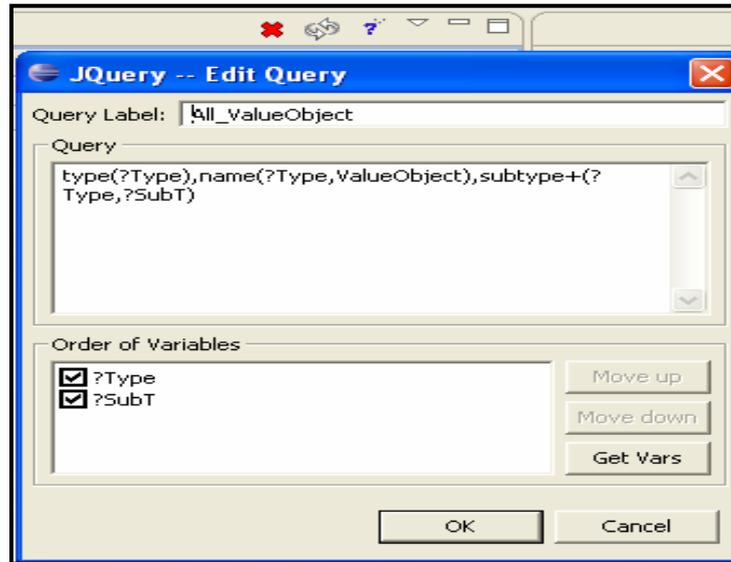
Figure 4.3: An early exploration of the AZ-VUB application code

Example 2:

In this example, we want to focus more specifically on the code to be extracted. We had already identified the "be.azvub.j2ee.arch.util.logging" package as containing most of the functionality of the logging concern we were interested in. We needed to find out how to concretely use of this concern. Figure 4.4 shows how we discovered the caller method for all methods (returning void) of all classes of the logging package.

The query typed in the example is a set of sub-queries combined together incrementally to give the concrete output. In this query we uses the filter capability of JQuery by using rename condition to filter the result to display the classes of "logging" package and the methods returning void.

This tool helps us at the starting of mining phase in viewing and understanding the structure of our case study and also helps us in querying about the seeds discovered by automated tools or about well-known crosscutting concern. Therefore using JQuery could be complement for using other tools.
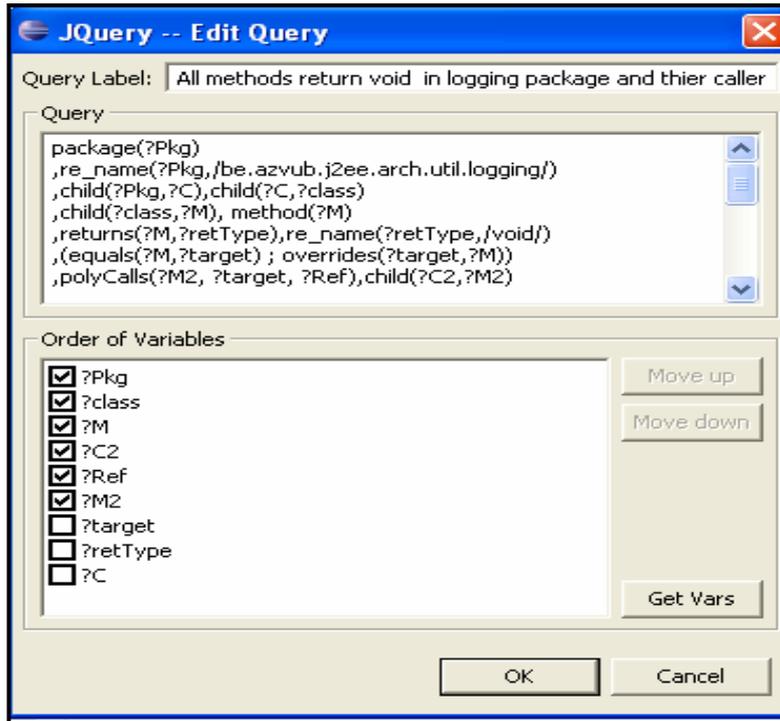
Figure 4.4: Exploring the usage of the logging concern

4.3.3.2 Applying FEAT Tool

FEAT is a tool by which we can build up concern graphs in an easy-to manage way through the Eclipse IDE. Concern graphs are collections of program elements of interest, such as classes, methods, and fields. FEAT also allows us to include some relationships of interest for each element in the concern graph, while excluding others. FEAT tool helps us in identifying and collecting the elements and relationships for identified concerns whose seeds discovered by the automated tools.

Using FEAT, we build concern graphs for the identified aspect candidate. The graph is a set of vertices and edges among the vertices. The vertex is presenting one of the program elements which can be class, method or field. The edges among vertices are presenting the relationship between the program elements constructed by using querying capability of FEAT tool. In FEAT we can query the source code using two categories of queries: fan-in and fan-out.

We will give an example explaining how FEAT tool aiding us to spotlight on the logging concern usage in AZ-VUB application. We start our exploration from ServerLogger class as starting point. ServerLogger is the major class in the server logging facility. ServerLogger provides file logger to the server environment. The ServerLogger instance can be retrieved by using getLogger("aModule").

To find out where the program locations use the ServerLogger, we make fan-in query to get all incoming call for the getLogger method. The query output is shown as tree containing all classes where the methods calling getLogger in order to log the message and the exceptions upon the ServerLogger object by using info(), warning(), error() or alarm() methods.

We also found FEAT very helpful simply for keeping track of the work done, and the concerns that we were looking at.

The ability of FEAT to save a concern and restore it later is very useful as a refresher to remind ourselves where we were in the process of refactoring.

With FEAT concern graphs, we can understand the code structure in abstractly way excluding the large details of the implementation. This abstraction captures the core of the relationships between different code elements, making it easier for us to focus on the concern. When required, elements can be mapped to source code to access the details.

Next we will present some explanation for certain of candidate aspects identified through exploring for the base code of the AZ-VUB application. .

Precondition Checking Concern

Another type of concern we discovered while exploring the code manually was precondition checking. The precondition checking often requires duplicated code if the conditions are common to many methods. In our case study many classes implementing javax.servlet.Filter have a doFilter method which expects a request as input parameter. We observed that there are frame filters classes which test the request parameter that should be have attribute for frame filter type. The parameter checks occur at the beginning of the method .

Another instance of the concern that we also discovered while exploring the code were null checks and verifying checks testing. We found four methods in a utility class named "ContextParams" in the package "medication". These methods use a precondition statement at their beginning. The methods are used in generating URL fragment containing one or more parameters for adding it to an HTTP request. The founded precondition statement is used to exclude particular parameters to be not used in forming the request. Therefore there is a set named "exclude" containing these parameters that should be excluded.

4.4 Evaluation

In this section we give an evaluation view for the tools used in the mining experiments and for the results of mining process.

We will try at first to highlight some observations related to technical issues of the used tools. We do our mining experiments by computer with Intel Pentium processor 1.86 GHz and 512 MB of RAM. When we applied those tools on the AZ-VUB application, we observed that FINT (version 0.3) required long time around 15 minutes to parse the application and compute fan-in values, while Prism required 2 minutes to show the advisor. We also observe that JQuery creates a large number of small files on a hard disk, using them as a sort of database called fact-base. Additionally JQuery keeps a subset of the fact-base in memory at all times. If you have a large code base JQuery requires more memory to keep that subset. This all leads to huge amounts of memory usage, which can end up with Java"OutOfMemoryError" error and is rather an unproductive resemblance of a database.

We observe that FINT tool (version 0.3) has potential to analyze the user-defined types yielding the fan-in value for each defined method that is called within many places. We also observe that FINT can't deal with the methods of pre-defined types used in the application, if their source code is not available in the application. The limitation of FINT in analyzing the entities of these pre-defined types used in the application could make the FINT to be not able to find out all the methods scattered throughout the system.

Prism has the capability to analyze the both types, user-defined types, and predefined types yielding advisors showing views of the methods or the types ranked according to its scattering degree. Therefore several aspect are discovered whose elements related to pre-defined types by Prism. Prism also allows us to make queries over the base code of the system. Prism misses the code elements whose low scattering degree. We observe that both FINT and Prism are efficient tools in discovering the dynamic crosscutting concern related to the method calls scattered throughout the system. The large results of the both tools need effort from us to analyze and indicate which the entities of the results could be good aspect candidates.

The FEAT tool and the JQuery tool are fine tools in discovering and collecting the entities of the concern whose well-known seeds. Therefore to make the mining process to be effective, using those tools require starting points to begin the mining activity. These tools are helpful in discovering the static crosscutting concerns related to the class hierarchy of the system. Those tools also help us in finding more complete concerns based on initial seeds identified because the concern identified by FINT and Prism tools is larger than just the methods calls, relating to setting up appropriate objects and checking relevant conditions.

Our aspect analysis results indicate that modularity of AZ-VUB application design is greatly limited by the wide existence of tangled logic. The most of the crosscutting concerns discovered in our mining activity can be sorted into one category namely consistent behavior. The category includes concerns related to transaction control mechanism, notifying listeners, exception handling, precondition checking, exception wrapping, clock setting, and events logging.

We will go further steps in refactoring some of these concerns by extracting them into aspect using AOP constructs, so the next chapter presents these steps in details.

Chapter 5

Introducing Aspects in AZ-VUB Case Study

We will go one step further; we will factor out a number of the crosscutting concerns identified in the mining process discussed in the previous chapter and re-implement them as aspects. Therefore, this chapter presents in detail the AOP refactoring process applied on the AZ-VUB application. We could not test the refactored code since it is incomplete.

5.1 Applying AOP refactoring to AZ-VUB application

The idea of refactoring to aspects is to modularize concerns that are candidate aspects, identified in the aspect mining phase. For modularizing field and method declarations that are originally scattered, inter-type declarations are used. They effectively take out the declarations from the classes and add them in a modularized aspect. For other kinds of refactorings, we use advice with pointcuts which select identified execution points to move scattered code into an aspect. We utilized these techniques in the refactorings described in the next sections. So the next sections discuss and present the refactoring of some of the crosscutting concerns identified in the AZ-VUB application through the mining process presented in the previous chapter.

5.2 Extracting the Notifying Listener Concern

In the aspect mining process, we detected the notification concern in the CarePlanTableModel class. Most of the methods of that class are tangled with a call to the method named "fireTableStructureChanged", which notifies the listener of the care plan table when the state of the table is changed. The notifying listener concern is comprised of 3 types of code sections: one field representing the listener reference, methods implementing the logic of the concern and code fragments which are calls of the notifying concern's methods. We dealt manually with each of these in turn, as described in *Extract Feature Into Aspect* [MF04]. Moving fields and methods was

straightforward and done according to *Move Field From Class To Inter-type Declaration* and *Move Method From Class To Inter-type Declaration* [MF04]. Extracting the calls of the fireTableStructureChanged method into an aspect was done by isolating these calls from the source code of the core concern. Therefore we defined pointcuts capturing join points at which the state of the table is changed and the notifying action is triggered in turn. We defined "after advice" for those pointcuts to trigger a notify method after the execution of the methods changing the care table state.

The implemented aspect in which we put the elements related to the notifying concern moving it from the CarePlanTableModel class into the aspect and shows the after advice encapsulating the calls of fireTableStructure- Changed method.

Two methods named "makeAndInserCarePlanEntry" are a special case, in which the triggering of the notify method is conditional with incrementing the column count of the care table after updating the table; therefore to refactor that case, we defined "around advice". The advice checks the input parameter of type BasicActivityVO that should be not null, and stores the value of the column count before proceeding the method to be checked later in order to trigger the notifying action.

The aspect implementation uses pointcut in enumerative style to express guaranteed join points at which the aspect crosscut the base code. The main problem associated with this style of the pointcut is high coupling between the aspect and the base program so naive modifications to the program could make the pointcut to loss the information of its intended join points. Another style of pointcut definition is a pattern-based pointcut that could be used instead of enumeration pointcut to reduce the coupling. The definition of based-pattern pointcut relies on the naming convention to arrange the code in name patterns so expressing pattern-based pointcut in this case would be hard and matched methods accidentally complying with this name patterns. With this refactroing for the notifying concern, we could improve the readability and the reusability of the CarePlanTableModel class. By using the aspects, it is possible to plug or unplug functionality of the notifying action.

5.3 Extracting the Transaction Control Concern

The Container of EJB objects manages transactions performed through the execution of business methods. The Container management is based on transactional properties

for the business methods declared in a separate file: the XML deployment descriptor. The deployment descriptor declares the transaction requirements for each business method. For example, setting the "*Required*" property for a method means that the method must be executed within the scope of a transaction, and if needed, a new transaction will start, while specifying "*Mandatory*" means that the method invocation will fail if not executed within a transaction scope [Fab04].

If a transaction is created upon execution of a method, the Container will commit or rollback this transaction when the method ends. The decision to rollback the transaction is primarily based on the type of the exceptions thrown.

If the method throws a system exception, such as EJBException and this exception is not caught within the method, then the method will terminate and the container will begin a rollback. The application-defined exception will not bring about the same effect. In this case, calling setRollbackOnly() method on the Container is needed before throwing the exception. At any point the method can call setRollbackOnly(), which will indicate that a rollback is to be performed when the method exits.

Therefore the "setRollbackOnly" method call is scattering throughout the business methods of the EJB objects performing a secondary concern crosscutting the core concern of these methods. This concern was detected by the mining activity; for example we found 40 methods performing transactional actions in the OrderEntrySystemBean class, crosscut by that concern controlling the transaction execution.

We refactored this concern for the OrderEntrySystemBean class in an aspect handling the calls to the "setRollbackOnly" operation. We started with identifying all transactional methods needing rollback actions to be taken when an exception is thrown through its execution.

We specify these methods first in order to make appropriate join points capturing the execution of these methods. We used a pattern-based pointcuts getting 11 pointcuts for those methods. We defined "around advice" in the aspect to be triggered at these pointcuts. Inside the advice we used try-catch blocks to trap the exception thrown when the method proceeds. Inside the catch block, the "setRollbackOnly" method is invoked on the Container object, after that the exception is re-thrown. Due to the AspectJ lack of a pointcut designator triggering before the exception throwing, we preferred to use "around advice" instead of using "after throwing" advice in this case

because the "setRollbackOnly" action should be taken before the exception throwing to save the behavior of the original code before the refactoring process.

The pointcut definitions seem not meaningful and understandable. Due to the large number of the pointcuts matching signature of the methods, the aspect appears more complex and not easy to understand and reuse. To overcome this problem and make the aspect more legible, we propose to define an abstract aspect named AbstractTransactionControlAspect.

The abstract aspect performs the transaction control logic necessary for executing the transactional methods. This logic is expressed in the around():transactionActivities() advice. The aspect presents two abstract constructs that must be overridden when we implement a concrete transaction control aspect

- public abstract pointcut transactionActivities();expresses the pointcut where the advice must be applied. The pointcut must be a method call.
- public abstract SessionContext getSessionContext (Object o);must return an instance of a SessionContext presenting the Container component of the facade session class.

The benefit of the abstract aspect that it can be reused for any session faced class performing transactional methods.

To apply this abstract aspect on the OrderEntrySystemBean class we define an interface named TransactionalMethodsInterface containing the signatures of all transactional methods of the OrderEntrySystemBean class.

We then reuse the abstract aspect by implementing a concrete aspect extending the abstract aspect. In the concrete aspect shown the declare parents static crosscutting construct was used to make the facade class, which contains all transactional methods of the order entry system, implement the Transactional- MethodsInterface interface. Then, we defined a concrete transactionActivities pointcut to identify the transactional methods of the order entry system. The pointcut matches the execution of all methods defined by the TransactionalMethodsInterface interface.

Finally, we define the concrete getSessionContext method to return the container instance for the target object of the transactional methods. So casting process was proceeded for inside the method to convert the type of the argument which presents the target object of the captured method (join point).

With this approach, we can improve the readability and the reusability of the aspect; also we guarantee preservation of original behavior for the refactored code by defining one pointcut capturing the transactional methods and replacing the large number pattern-based pointcuts that could be matched methods accidentally or lost required methods. If we need to plug or unplug functionality of the transaction control for particular methods, we just update the interface by either adding a new signature or remove the existing signature. In this version of the aspect does not suffer from the same problems as those with the original version.

Also With this approach, the aspect is not directly dependent on the transactional methods signatures, but the auxiliary TransactionalMethodsInterface interface is totally dependent on them so any changes to these methods require update to their corresponding in the interface.

5.4 Extracting the Exception Handling Concern

Another concern we discovered by mining the system was exception handling. There are eleven classes in AZ-VUB application, implementing the Action interface by encapsulating the EJB functionality actions. So each of these classes is implementing an execute method triggering the appropriate EJB functionality whenever an HTTP request invokes the corresponding URL. We observed that each execute method is crosscut by a concern handling the thrown exceptions. All execute methods have one way to handle the exceptions encountered during execution of their core logic so the exception handling code is duplicated in all the execute methods.

To isolate the exception handling concern from the core concern of these methods, we implemented an aspect, in which we defined a pointcut capturing the executions of the execute methods for the classes sub-typing from the Action type, and also we defined "around advice" for that pointcut. In the advice, proceeding for the captured executed method is done in try-catch block trapping the thrown exception.

The execute methods after the isolating the exception handling concern that implemented in the aspect. The benefits realized from this refactoring are localizing exception handling code in one place, and reducing the duplicated code.

5.5 Extracting the Persistence Concern

ResultSetWrapper class warps the java.sql.ResultSet object used as a delegate object to get data which is acquired by executing a statement querying the database. All the

getter methods, in the ResultSetWrapper class, are crosscut by a call to the storeField method that stores the resulting data in a hash table. The hash table is considered as a temporary store to be used later in optimizing the creating of HTML table displaying these data.

To refactor this crosscutting concern, we implement an aspect in which the pointcuts capture the execution of the getter methods. We define "after-returning" advice for that pointcut in which we can access the returned value and store this value using the storeField method.

There are two versions of the storeField method, one for storing the value in the hash table using column name as a key, and the other for storing the value using column the index as a key. So the aspect handles the two cases by defining two pointcuts for the getter methods one for getter methods that accepts an integer argument used as the column index and the other pointcut for the getter method that accepts argument used as the column name. There two advices for each pointcut. Each advice invokes the appropriate storeField method with two arguments. The first argument is used as a key in the hash table and the second is the value that would be stored in the table.

5.6 Extracting the Precondition Checking Concern

Another type of concern we discovered while exploring the code manually was precondition checking. The precondition checking concern often requires duplicated code if the conditions are common to many methods. In the AZ-VUB application many frame filter classes implementing javax.servlet.Filter interface have a doFilter method which examines the input parameter named "request" that should have attribute of the frame filter type. The parameter check occurs at the beginning of the method.

With aspect-oriented techniques, we can extract such contract checks into a separate aspect. In the aspect, we define one pointcut as "executions of the doFilter methods of filter classes whose name is ending with "Frame" " and we use an "around advice" to check the precondition before proceeding the method.

Another precondition concern is detected in "ContextParams" class where there are four methods that enforce preconditions at the beginning of each method. The condition testing the method parameters is duplicated in all of these methods. We refactored this concern by implementing an aspect using a pointcut capturing the executions of these method whose name starts with "add" and ends with "Parm".

An around advice was used to defer the method execution in order to check its arguments before proceeding the method. The captured methods signatures have two forms.

The arguments needed in the test are "paramName" and "exclude" that are always at the second and the last position respectively. So to handle this situation, we use the reflective method getArgs() allowing us to access these arguments of the captured method. After getting the argument values, the advice checks these values in a condition then the captured method is proceeded, if the condition is true.

5.7 Extracting the Exception Wrapping Concern

Exception wrapping is a crosscutting concern detected through the mining process. We observe that most of the business methods in the case study catch exceptions thrown by the underlying implementation and re-throws application-specific exceptions (J2EEArchException).

Applying this type of exception handling mechanism requires one or more try/catch block in each method. In each of the catch blocks, a new application-specific exception wrapping the caught exception is created and re-thrown. We observe that one method of the SessionBean class uses multiple try/catch blocks to check the exception thrown, and then wraps the exception in a new application-specific exception. These try/catch blocks increase the code size and makes it more complex.

With AOP, we refactored this concern by implementing an aspect, in which the checked exceptions are declared soft [Lad03]. Each declare soft statement causes any exception of the specified types (ClassNotFoundException, NoSuchMethodException, RemoveException, EJBException, IllegalAccess-Exception, InvocationTargetException) thrown from the executions of the methods captured by the specified pointcut to be treated as a runtime exception. This way the exceptions will be wrapped in an unchecked exception (org.aspectj.SoftException) when thrown. An after-throwing advice is then used to catch any SoftException thrown.

We observe the developer uses static strings in wrapping the thrown exceptions to provide proper error information for the actual exception. These static strings make it more difficult to create an AOP implementation; therefore there is a specific limitation in figuring out the static strings to give the actual trace of the exception. However, the tracing information acquired in the AOP system is limited to the

information provided by the joinpoint (name of the method, arguments, class name, and so on). To output the same information as the OOP implementation, we require building several quite complex pointcuts to define what we want to display. We therefore partly lost the benefits of using AOP.

But after extracting exception wrapping concern, refactoring the exception wrapping concern cleans up business logic that is not tangled with wrapping anymore. This not only leads to a reduction in code size in the refactored classes, it also improves readability and evolvability of the business logic.

5.8 Extracting the ServiceLocator Concern

The ServiceLocator is implemented by the GoF Singleton pattern and has a private constructor and a factory method (getInstance) whose high fan-in was detected in the mining activity. Hanneman et al. [HK02] presented in their research how a plain old java object (POJO) can be turned into a Singleton by using AOP mechanisms. The Service Locator can be instantiated like a POJO using the new constructor instead of using a factory method like getInstance. We can refactor the singleton class by an aspect. The pointcut of the aspect intercepts all calls of the class constructor and provides around advice. The advice creates an instance of the class (if it is not created before) and returns the instance. Other refactorings can be applied to the ServiceLocator class before applying the aspect: convert the accessor modifier of the constructor from private to public and remove getInstance method. These refactorings will allow other classes to create an instance of the singleton using the singleton constructor instead of using the factory method getInstance.

However, hiding the singleton nature of ServiceLocator can lead to some confutation among J2EE developers as is mentioned in [MPY04]. A factory method makes it clear that the Service Locator is a singleton but the new constructor does not.

5.9 Conclusion

In this chapter, we show aspects for some crosscutting concerns that were detected by mining the AZ-VUB application. We also discussed the implementation of these aspects that achieve number of improvements over the existing code. The benefits gained through the refactoring process are getting cleaner modularizations by

encapsulating the crosscutting concern within separate modules, giving cleaner code that are often easier to read and maintain. There are some disadvantages causing challenges in applying AOP. Some of these challenges are not always easy to write cleaner aspect and not easy to create robust pattern-based pointcut. If the aspect is not clear and less readable, the aspect needs to refactor its code.

Chapter 6
Road Map

In this chapter we outline our experiences gained from using AOP technology in migration an industrial application into aspects. Therefore we explain the lessons we learned from migrating an application to aspects. Also we illustrate the pitfalls that come with the migration process. We think that these practices can help others in similar situations to improve the effectiveness of software maintenance.

6.1 What have we learned?

In the previous two chapters, we discussed two phases of using AOP in migration an enterprise Java application into an application with aspects. Through the first phase, some of crosscutting concerns are detected by mining the system. These crosscutting concerns are refactored to aspects in the second phase. In this section, we try to show a general view, summarizing the learned lessons and the pitfalls during the migration process.

The First Lesson Learned: Extracting the crosscutting concern from an existing application requires from the aspect developer some effort.

Extracting the crosscutting concerns from an existing application requires from the aspect developer some effort in:

- Understanding the application target code.
- Choosing the proper aspect mining tools that are used in detecting the crosscutting concern automatically and learning how applying these tools on the target source code.
- Analyzing the results yielded from these tools to select aspect candidates.
- Extracting the aspect candidates into aspect using one of AOP approaches.

All of the above situations could face any developer wanting to transfer an already completed application into application using AOP technology. These activities are time-consuming activities. Although tool support exists, we still need to invest a lot of time.

There are assumed advantages gained form using the new AOP technology that outweighs the difficulties and the required lead-time in learning the AOP technology. With aspects, application code is reduced; the code is more easily re-used and evolved; the code is easier to understand; etc. However, AOP can be difficult to apply to an already completed project so lot of effort needed to understand the target source code and to manipulate with it. If AOP is to be used, this should be known at the design phase and applied by the core developer. Initially, we tried to illustrate the use of aspects by modifying the original AZ-VUB Java source code. However, we soon found out that this is not the best practice. It is very difficult to extract all the code that belongs to a particular aspect into a single place, because one has to be very familiar with and go through all the code of the application.

For example in the notification concern in the CarePlanTableModel class, we found our self restricted to extract some elements of the concern, because we have not more knowledge about the appropriate events at which calls to concern's methods should be taken. One of these methods is a "refreshCurrent" method that is called to make the model's listener to refresh its current-displayed part. There are situations of the calls to the "refreshCurrent" method after the calls to the notifying method "fireTableStructureChanged" directly, but there are others situation the calls are different.

The Second Lesson Learned: The joint point model of AspectJ is too restricted for the purpose of the refactorings we did.

When we implement aspects using AspectJ, we learned that some limitations of AspectJ (Version 1.2.1) made it difficult to handle certain kinds of problems. For example AspectJ does not provide support to access local variables in the join point. AspectJ allows advice to reference variables related to a joinpoint. Such variables are:

- The object making the call (this).
- The object receiving the call (target).
- The values of the method's parameters.
- The returning value of the method.

The advice in the aspects does not have access to local variables around a join point, except for the above mentioned variables.

For example, if we have code as shown in listing 6.1 and we want to extract the call of the method m1 (see listing 6.1,line 7 and 8) from the method m2 into aspect by making advice triggered after the execution of m2. The advice calls the method m1 (see listing 6.2).

```
1   Class x {
2   void m1(){
3   …}
4   void m2(){
5   int  var =4 ;
6   var = var + doSomeThing();
7   if(var ==someThing)
8   m1();
9      }
10  }
```
Listing 6.1: Example Java code

```
1   Aspect AspectX {
2   after(X x): execution (void X.m2())&&target(x){
3   //the condtion (if(var ==someThing))must be implemented here
4        x.m1()
5           }
6   }
```
Listing 6.2: Difficulty in using local variable in the Aspect

As seen in method m2, the call of m1 is called at the end of method m2, so it is possible to extract this call by using after-advice but there is a difficulty that the call of m1 is conditional with the test respected to local variable of m2. This difficulty will limit such a refactoring of extracting the method call. The solution in this case is that the local variable of the method could be converted into a field of class to be accessible to advice.

The Third Lesson Learned: there is difficulty in extraction heterogeneous crosscutting concerns:

We found crosscutting concerns implemented in a heterogeneous manner. The crosscutting concern is scattered through different places and applied in varying ways. For instance exception wrapping is a crosscutting concern scattered throughout the AZ-VUB application. The intent of that concern is wrapping the different exceptions thrown in the system to provide extra information to the user by adding new clear messages.

The heterogeneousity comes from using different constructors to create the wrapper exception and also comes from using the static strings (messages) passed to the

constructors to present the information that used by the user to trace the exception. This heterogeneous code brings difficulties if we need to refractor this concern.

In spite of existence of these difficulties, we were able to understand much better how the legacy application can be migrated into aspects maintaining its structure.

To solve this problem, multiple pointcuts and advices should be defined to handle the different situation of the concern and provide the AOP implementation similar to OOP. These pointcuts and advices will be more complex causing aspect to be illegible. Our solution uses one constructor for creating the wrapper exception and use the information provided by the join point and the thrown exception to be passed to the constructor. Therefore our implemented aspect does not provide static strings similar to OOP implementation.

Finally, we were able to build up an understanding of the aspect code, and how it applies to the various points in the existing code base. We were able to consistently improve the quality of the pointcuts and aspects that we wrote, both in terms of the places where they apply, and the conciseness of the aspect. Building an aspect containing abstract pointcut can make it to be reused in different concrete situations. Also choosing the proper events (join points) will increase the quality of the aspect in providing consistent behaviors and reducing some side effects rose when incorrect pointcuts are selected.

6.2 How to migrate to aspects in general

Migrating an existing system into an equivalent aspect-oriented version is a process performed in several steps. The steps are divided in two phases:

The first phase is the aspect mining phase including activities aiming to detect the inelegant-designed sections (bad code smells) of the application that can be handled by AOP approaches. These application parts are elements implementing secondary roles crosscutting the core concerns of the system. The aspect mining process is used to discover these crosscutting concerns that reflect maintenance and evolution problems.

There are several aspect mining tools that can be used in detecting the crosscutting concerns. In chapter 3(section 3.1), we discussed in detail these tools, such as tools are Fan-in tool, Prism, FEAT, JQuery, etc.

The results gained after applying the aspect mining tools are application sections that might be seeds of the crosscutting concerns. Therefore these results require more analysis to spotlight the real and whole structure of the crosscutting concerns. This is not an easy task to distinct between the positive seeds and the negative seeds so several challenges come with this task. An important problem is that the crosscutting concerns are difficult to understand, because their implementation can be scattered over many different software components. The automated mining tools provide an overview of the source-code elements that play a role in a particular crosscutting concern so some effort needed to improve the understandability of the concern in particular and of the software in general.

The second phase after aspect mining is aspect refactoring. The aspect refactorings are transformations of the internal structure of the application extracting the identified crosscutting concerns into aspects. There are several AOP languages that can be used in this phase. In chapter 2 we discussed in detail these languages, such of theses languages AsepctJ, JAsCo, HyperJ, etc.

To refactor the crosscutting concern using aspects, there are AOP refactoring mechanisms that can be applied to the application code to extract these concerns. In chapter 3 (section 3.2) we explained certain of these mechanisms.

After extracting the crosscutting concerns into aspects, the behavior of the refactored application must be maintained. Therefore testing and evaluation the behavior of the application should be achieved to ensure that the refactoring process did not introduce bugs.

6.3 What are the pitfalls?

Based on our experiences, we put found some pitfalls of applying aspects to an already existing application. We also explain some of the difficulties that others may face when using aspect technology in similar situation.

At first we explain in particular the pitfalls in the AZ-VUB case study. The AZ-VUB case is J2EE platform-based application that comprises 37 packages including 408 types. These types contain around 4535 methods including approximately 7622 LOCm. The large code size brings difficulties in understanding the system and making a detailed look

at the case architecture. In addition there are also difficulties in applying the mining tools and analyzing the results yielded form applying these tools. The lack of information about the system and enough documentation make us weary of changing any thing.

The application is based on EJB and utilizes layered architecture using established EJB design patterns including data value objects, session facade, service locator, and business delegates. Although J2EE has advantages, it adds a layer of complexity to the application especially session beans that bring more complexity to the code and require more work to maintain and evolve. Our case study is part of the AZ-VUB working system so we have not complete source code of the system. Also we can not run the code so there are more difficulties we faced.

Pitfalls involved in identifying crosscutting concerns:

An application migration into aspects needs developers to pay careful attention on extra considerations, such as being able to identify and understand the crosscutting concerns correctly. Extracting the crosscutting concerns correctly depends on the identification process for those concerns therefore the developer at this phase must be provided with the needed information about the target system.

This information could be classified into two types: static information and dynamic information. Looking to the source code and what is included in the comments is a way to get the static information describing the software structure; the dynamic information can be obtained by running the software to get more information about the behavior of the software. This information helps the aspect developer to identify aspectual requirements and their relationships with other requirements.

One of the main problems we faced at the start of our experiment was understanding the target source code because there is not enough documentation for the target system, so we are limited to studying the source code and comments because we could not run the code. This issue can be time consuming and unrealistic for complex application. The problem of mis-understanding and analyzing the entire information and requirements of the application's concerns maybe create certain side effects on identification of the whole and the correct structure of the crosscutting concerns and refactoring these concerns; for example extracting incorrect entities related to a particular concern will introduce

difficulties in the refactoring process (where and when the refactorings for these entities are applied).

While exploring the concerns, we definitely came to understand the code much better than at the start. As we were building up some of the concerns and performing the refactorings, we discovered more about the purpose of the classes and methods that we were looking at.

There is another important issue in choosing and using the proper tools used in mining of the crosscutting concern. There exist different mining tools; each one has advantages and drawbacks. In our experiment we used four tools, 2 automated tool (FINT and Prism) and 2 explorative tools (FEAT and JQuery). FEAT is very effective mining tool. FEAT allows us to figure out code locations referencing some method or field. These references could be replaced with an aspect to perform the aspect-oriented refactoring. The automated tools analyze the application's entities yielding large list of results. So to handle all these results by analyzing all of them carefully, you need more time and attention to choose the proper aspect candidate (crosscutting concern) and identify the code elements related to that concern.

Most of mining tools used in our experiment are focusing on the crosscutting concern resulting from scattered method calls, so through our manual exploration for the code base, we observe that there are a lot of duplicated code sections. These duplicated sections might be forming crosscutting concerns that can be handled by AOP technology. Therefore analysis the code using tools discovering these duplicated sections will be worthwhile means to identify these crosscutting concerns and give the aspect developer more insight into crosscutting concerns that might not be visible by just exploring the code manually. We think that augmenting the mining process with clone detection tools might be a productive approach for aspect mining. For example, the exception handling concern often requires similar pieces of code to handle the similar errors which could be refactored into an aspect. So using the clone detection tools for detecting duplicated code may be beneficial for aspect identification.

Pitfalls involved in refactoring crosscutting concerns:

There are also pitfalls arising in refactoring the crosscutting concerns. After identifying the crosscutting concern and finding all the elements related to it, you must correctly select the proper AOP constructs to manipulate that concern so this is one of the critical issues that the aspect developer must pay care to. For example, making a mistake in defining pointcuts to capture joinpoints for scattered calls for a particulate method by losing or adding joinpoints maybe introduces incorrect behavior in the application. So you must be careful in defining a correct pointcut.

Choosing an appropriate manner for extracting the crosscutting concern is surrounded by a number of pitfalls. For example, choosing incorrect joinpoint at which the advice's code is triggered. This situation faced us in choosing proper event for triggering advice's code to enforce EJB container to do the rollback action before exception throwing through the transactional methods. We seen to use around advice instated of using after-throwing advice because of restriction of the AspectJ language to trigger before throwing exception. In the around advice the transactional method is proceeded in try-catch blocks to trap the thrown exception and enforce the rollback action in the catch block then re-throw the trapped exception. This alternative solution is made to avoid any bugs that maybe introduced from using after-throwing advice.

The validation process must be executed after any refactoring process to ensure if the performed refactorings introduce bugs or not.

In certain situations, AOP implementations have shortcomings or limitations in giving implementation for a specific problem similar to its OOP implementation. For example, in chapter 5 (section 5.7), we demonstrate how we can refactor the exception wrapping concern. We observe that in the OOP implementation, the developer uses static strings in wrapping the thrown exceptions to provide proper error information needed for tracing the actual exceptions. In our AOP implementation for that concern, we found our self restricted for providing this static information similar to OOP implementation therefore there is shortcoming in AOP implementation in giving the static strings tracing the thrown exceptions. However, the tracing information acquired in the AOP implementation is limited to the information provided by the joinpoint (name of the method, arguments, class name, and so on).

6.4 Conclusion

Using AOP technology allows the duplicated code to be identified and handled. AOP makes the target application better understanding and improves its evolvability.

In using AOP, a lot of effort and knowledge are needed form the developer for extracting the crosscutting concern. There are also a number of difficulties in using AOP for the refactoring process. One of these difficulties is involved in creating a robust pointcut and advice. The mismatched of AspectJ join point model brings some of these difficulties in refactoring the crosscutting concerns.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we have experiment on an industrial application. The main goal of our experiment is transforming industrial application from OO to AO application. We applied these transforming in two phases: Aspect Mining and Aspect Refactoring. In aspect mining phase there are several tools used to detect crosscutting concerns in the code. In our experiment we used Fan-In, Prism, FEAT, and JQuery tools to detect crosscutting concerns in the code. We noted each tools have a numbers of advantages and disadvantages. So, we conclude there is not to date a single tool can detect each crosscutting concerns in the code. Therefore, we need using several tools and detect manually sometime in order to try detecting most crosscutting concerns in the code. In aspect refactoring step there are several aspect-oriented languages we used to actually extracting the discovered crosscutting concern into real aspects in the code. We have known how aspect-oriented languages can clear software code from these crosscutting concerns yielding fine software modularity. We have seen different aspect-oriented mechanisms that provide additional flexibility in modularization to capture the location and behavior of crosscutting concerns, resulting in to the highest degree evolved separation of concerns. In our experiment we used AspectJ to refactor our case study. We have achieved a number of evolvements over the existing code, like: easier to understand, maintain, and reuse. And also reduce the code duplication. Finally, in our experiment we noted some pitfalls we faced:

- **Pitfalls involved in identifying crosscutting concerns.**
- **Pitfalls involved in refactoring crosscutting concerns**.

7.2 Future Work

We would like to use another aspect mining tools and aspect-oriented languages in our experiment and in other applications. Therefore, for trying detect a numbers of extra advantages and disadvantages of the mining tools and aspect-oriented languages. We noted in our experiment there is not to date a single tool can detecting each crosscutting concerns in the code. Therefore, we will try to develop mining tool which can solve problems which found in previous tools. Finally, in our experiment we noted some aspects are depending on the type of application. Therefore, we would like to implement more generic aspects that can be reused in several applications.

Reference:

[AB]   AspectBrowser   for   Eclipse,
            http://www.cs.ucsd.edu/users/wgg/Software/AB/.

[ajd]   AspectJ Team, The AspectJTM 5 Development Kit Developers's Notebook,
            http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-
            home/doc/ajdk15notebook/.

[alp]   Alpha   project,   http://www.st.informatik.tu-
            darmstadt.de/static/pages/projects/alpha/.

[amt]   TheAspect Mining Tool, http://www.cs.ubc.ca/labs/spl/projects/amt.html.

[asp]   AspectJ language, http://www.eclipse.org/aspectj.

[aspe]   Xerox PARC, USA, AspectJ Home Page, http://aspectj.org/.

[BA04]   L. Bergmans and M. Aksit. Principles and Design Rationale of Composition
            Filters. Aspect-Oriented Software Development, 2004.

[BB02]   E. Burd, and J. Bailey. Evaluating Clone Detection Tools for Use during
            Preventative Maintenance. SCAM 2002.

[BCC05]   K. Berg, J. Conejero, and R.Chitchyan. AOSD Ontology 1.0 -Public
            Ontology of Aspect-Orientation 27 May 2005.

[BCH+05]   D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P.Tonella. Automated
            refactoring of object-oriented code into aspects, 2005.

[BDET04]   M. Bruntink, A. Deursen, R. Engelen, and T. Tourwé. An Evaluation of
            Clone Detection Techniques for Identifying Crosscutting Concerns. In
            Proceedings of the International Conference on Software Maintenance
            (ICSM).   IEEE   Computer   Society,   2004.
            http://citeseer.ist.psu.edu/article/bruntink04evaluation.html.

[BH05]   J. Brichau and M. H. (editors). Survey of aspect-oriented languages and
            execution models. Tech. Rep. AOSD-Europe-VUB-01, AOSD-Europe, May
            2005.

[blu]   Find   Sun   Microsystems'   J2EE   BluePrints   design   patterns,
            http://java.sun.com/blueprints/.

[Bre04]   S. Breu. Towards hybrid aspect mining: Static extensions to dynamic aspect
            mining. In 1st Workshop on Aspect Reverse Engineering, 2004.

[cae]   CaesarJ, Retrieved on 04/04/2006, http://caesarj.org/.

[CCHW04]   A. Colyer, A. Clement, G. Harley, and M. Webster. Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. December 2004.

[CMM+05]   M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. A Qualitative Comparison of Three Aspect Mining Techniques. Program Comprehension, 2005. IWPC 2005. Proceedings. 13th International Workshop on.

[com]   Composition Filters, http://trese.cs.utwente.nl/oldhtml/composition_filters/.

[dyn]   Dynamo - Dynamic Aspect Mining Tool, http://star.itc.it/dynamo/.

[ecl]   Eclipse Project,   http://www.eclipse.org/.

[ESS92]   S. Eick, J. Steffen, and E. Summer. 1992, Seesoft - A Tool For Visualizing Line Oriented Software Statistics, IEEE TSE, vol. 18, no. 11, pp. 957-968, November 1992.

[EV04]   A. Eisenberg, and K. Volder. JQuery: finding your way through tangled code. 2004.

[Fab04]   J. Fabry. Transaction management in EJBs: Better separation of concerns with AOP. In Y. Coady and D. Lorenz, editors, Proc. of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Victoria, Canada, 2004. University of Victoria.

[Fow00]   M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2000.

[GK05]   K. Gybels, and A. Kellens. Experiences with Identifying Aspects in Smalltalk Using 'Unique Methods'. January 10, 2005.

[Han05]   J. Hannemann. Role-based refactoring of crosscutting concerns. PhD thesis, University   of   British   Columbia,   BC,   Canada,   2005. http://www.cs.ubc.ca/~jan/.

[Han06]   J.   Hannemann.Aspect-Oriented   Refactoring: Classification   and Challenges.2006.

[HK02]   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, Boston, MA, 2002.

[HOU03]   S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In 4th Annual International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World (Net.ObjectDays), pages 19--35, Erfurt, Germany, Sept. 2003.

[HT05]   T. Hon, and M. Tkatchenko. Refactoring JQuery with AspectJ: an experience report. CPSC 511 Project Report. April 29, 2005.

[jas]   Jasco language, http://ssel.vub.ac.be/jasco.

[jav]   Java Technology. Java Platform, Enterprise Edition (Java EE). http://java.sun.com/javaee/.

[java]   Separate software concerns with aspect-oriented programming, http://www.javaworld.com/.

[JV03]   D. Janzen and K. Volder. Navigating and Querying Code Without Getting Lost. 2003.

[kgy]   Research: Aspect-Oriented Programming and CARMA. Retrieved on 04/04/2006. http://prog.vub.ac.be/~kgybels/.

[KHH+01]   G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. An Overview of AspectJ. 2001.

[KKI02]   T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7):645–670, July 2002.

[KM05]   A. Kellens, K. Mens. A Survey of Aspect Mining Tools and Techniques. June 30, 2005.

[Lad03]   R. Laddad. Aspect-oriented refactoring. www.theserverside.com, December2003.

[Mar]   M. Marin. Reasoning about assessing and improving the seed quality of a generative aspect mining technique. Software Evolution Research Lab Delft University of Technology.

[MDM04]   M. Marin, A. Deursen, and L. Moonen. Identifying aspects using fan-in analysis. In Proc. of the 11th IEEEWorking Conference on Reverse Engineering (WCRE 2004), Delft, The Netherlands, November 2004. IEEE Computer Society.

[MF04]   M. Monteiro, J. Fernandes.Object-to-Aspect Refactorings for Feature Extraction, 2004.

[MF05]   M. Monteiro , J. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. 2005.

[Mon04]   M. Monteiro. Catalogue of Refactorings for AspectJ, Technical Report UM-DI-GECSD-200402, Universidade do Minho, December 2004. Available at www.di.uminho.pt/~jmf/ PUBLI/papers/2004-TR-02.pdf.

[MPY04]   T. Murali, R. Pawlak, and H. Younessi. Applying aspect orientation to J2EE business tier patterns. In Y. Coady and D. Lorenz, editors, Proc. of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Victoria, Canada, 2004. University of Victoria.

[MT05]   K. Mens and T. Tourwé. Delving source-code with formal concept analysis. Elsevier Journal on Computer Languages, Systems & Structures, 2005.

[OT00]   H. Ossher and P. Tarr. "Multi-Dimensional Separation of Concerns and The Hyperspace Approach. Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000. http://citeseer.ist.psu.edu/ossher00multidimensional.html.

[pat]   Core J2EE Patterns , http://java.sun.com/blueprints/patterns/.

[Raj]   G. Raj. Enterprise JavaBeans. http://members.tripod.com/gsraj/ejb/chapter/.

[RM02]   M. Robillard, and G. Murphy. Capturing Concern Descriptions During Program Navigation. A position paper for the OOPSLA 2002 Workshop on Tool Support for Aspect Oriented Software Development.

[RoMu02]   M. Robillard and G. Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In Proceedings of the 24th international conference on Software engineering (ICSE), pages 406-416. ACM Press, 2002.

[RP06]   V. Ranganathan and A. Pareek. An Introduction to the Enterprise JavaBeans 3.0 Specification. http://dev2dev.bea.com/pub/a/2006/01/ejb-3.html.29/03/2006.

[SP05]   D. Shepherd and L. Pollock, "Aspects, Views, and Interfaces" Workshop on Linking Aspect Technology and Evolution at the International Conference on Aspect Oriented Software Development 2005.

[STP05]   D. Shepherd, T. Tourwé, and L. Pollock. Using Language Clues to Discover Crosscutting Concerns. 2005.

[SVJ03]   D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an Aspect-Oriented approach tailored for CBSD. In Proceedings of the second AOSD International Conference. Boston, USA, March 2003.

[TM04]   T. Tourwé and K. Mens. Mining aspectual views using formal concept analysis. In Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004). IEEE Computer Society, September 2004.

[ZJ04]   C. Zhang and H. Jacobsen. PRISM is Research In aSpect Mining. D.2.2 [Software Engineering]: Design Tools and Techniques Modules and interfaces. October 2004.