Detecting Bad Smells in AspectJ

Eduardo Kessler Piveta[1] **, Marcelo Hecht**[1] **,**
Marcelo Soares Pimenta[1] **, Roberto Tom Price**[1]

[1]Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Campus do Vale - Bloco IV
Bairro Agronomia - Porto Alegre - RS -Brasil
CEP 91501-970 Caixa Postal: 15064

epiveta@inf.ufrgs.br, mhecht@gmail.com, mpimenta@inf.ufrgs.br, tomprice@terra.com.br

Abstract. *This paper defines algorithms to automatically detect five types of bad smells that occur in aspect-oriented systems, more specifically those written using the AspectJ language. We provide a prototype implementation to evaluate the detection algorithms in a case study, where bad smells are detected in three well-known AspectJ systems.*

1. Introduction

Aspect-oriented software development improves the separation of concerns by providing abstraction and composition mechanisms that deal specifically with the modularization of crosscutting concerns [Kiczales et al. 1997]. The most common abstraction mechanisms are aspects, pointcuts, advice and inter-type declarations. Although the use of aspects might help in the modularization of crosscutting concerns, their use can introduce problems either particular to the use of aspects or similar to those found in objects, such as: pieces of code abandoned in a module and no longer used, code duplication and classes with too many or too few responsibilities.

These problems usually difficult reuse in several development process activities [Boehm and Sullivan 2000] and can be minimized by the identification of their symptoms and the removal of their causes. These symptoms (called bad smells by Fowler [Fowler et al. 2000]) may be seen as signs or warnings, indicating potential problems in the software [Elssamadisy and Schalliol 2002]. The problems could be removed or minimized by using appropriated refactorings to change the application design.

There are catalogs and descriptions of bad smells for object-oriented systems (such as [Fowler et al. 2000], [M.P. Monteiro 2005]), but their detection in aspect-oriented systems is still not enough explored. Monteiro and Fernandes discuss bad smells that arise in object-oriented systems [M.P. Monteiro 2005], indicating refactoring opportunities for code extraction from objects to aspects, without extensively discussing bad smells that occurs in aspect oriented systems. Piveta et al [Piveta et al. 2005] discuss several bad smells in the context of aspect-oriented systems. However, the authors do not provide mechanisms to automatically detect occurrences of those smells.

This paper focuses on automatic detection of bad smells in the context of the *AspectJ* language [Kiczales et al. 2001]. The main goal is to provide algorithms and a prototype implementation to detect five types of bad smells defined in [Piveta et al. 2005]:

anonymous pointcut definition, *large aspect*, *lazy aspect*, *feature envy* and *abstract method introduction*.

The remainder of this paper is structured as follows: in section 2., some mechanisms to bad smells detection are detailed as well as a implementation of an *AspectJ* bad smells detector, using an AST visitor based approach. In section 3., a case study with well-known aspect oriented systems is conducted. The chosen systems are: the *AspectJ* examples[1], *AspectJ Design Patterns*, [Hannemann and Kiczales 2002] and the *GlassBox Inspector*[2]. Section 4. describes related work and section 5. details final considerations.

2. Algorithms to Bad Smells Detection

Fowler [Fowler et al. 2000] presents bad smells as a way to identify problems in existing software artifacts. This is accomplished by suggesting possible symptoms that can appear in the artifacts, indicating areas that usually can be improved by refactoring. The use of refactoring techniques attacks the causes of those problems, causing the symptoms to be minimized or removed.

Some refactorings have been proposed to enable the code manipulation in aspect-oriented systems (such as: [Garcia et al. 2004], [Hanenberg et al. 2003], [Iwamoto and Zhao 2003], [Monteiro and Fernandes 2004], [M.P. Monteiro 2005]). These refactorings help to remove or minimize the occurrence of bad smells in aspect oriented code.

In this section, we describe how bad smells in aspect-oriented systems could be detected in *AspectJ* programs. A brief textual description, a more formal definition and algorithms are provided for each type of bad smell. A detailed discussion about the nature of bad smells in aspect oriented systems can be found in [Piveta et al. 2005].

2.1. Detecting Anonymous Pointcut Definitions

In *AspectJ*, as pieces of advice are not named, sometimes it is necessary to rely on the pointcut definition to have an idea of the affected points. The use of the pointcut definition predicate directly in an advice may reduce legibility and hide the predicate's intention. A name that clearly defines the pointcut intention could be defined and referenced by any advice or declare construction.

Definition 01 *Let $A = \{$call, execution, get, set, initialization, preinitialization, staticinitialization, handler, adviceexecution, within, withincode, cflow, cflowbelow, if$\}$ be the set representing all the primitive pointcuts in AspectJ that are not related to context exposure. Let B be the set of the tokens in a given pointcut definition. The pointcut definition is an anonymous pointcut definition if and only if the predicate $\forall b \in B \neg \exists a \in A | b = a$ is true.*

An implementation of a function to detect anonymous pointcut definition is showed in Listing 1. First, a set named *primitive*, containing all pointcuts not concerned with context exposure is created (line 2). After, the string *s* containing the pointcut predicate is divided into tokens (line 7), which are individually compared with the *primitive* set. If the set contains *s*, the method returns true, false otherwise.

[1] `http://www.eclipse.org/aspectj/doc/released/progguide/examples.html`
[2] `http://www.glassbox.com`

```
1  protected boolean isAnAnonymousPointcut(String s) {
2     Collection primitive = new ArrayList();
3     primitive.add("call");
4     primitive.add("execution");
5     ...
6     boolean temp = false;
7     String[] result = s.replace("("," ").replace(")"," ").split("\\s");
8     for (int x=0; x<result.length; x++)
9        if (primitive.contains(result[x])){
10          temp = true; break;
11       }
12    return temp;
13    }
14 }
```

Listing 1. A Java implementation of the *isAnAnonymousPointcut* function

The detection of anonymous pointcuts in *AspectJ* and *AJDT* could be done using a visitor, which visits advice declarations looking for the use of anonymous pointcuts. Listing 2 shows the implementation of such visitor. It visits all *AdviceDeclaration*, *AfterAdviceDeclaration*, *AroundAdviceDeclaration* and *BeforeAdviceDeclaration* nodes. Whenever the function *isAnAnonymousPointcut* returns true, a *BadSmellsEvent* instance is created to gather information about the bad smell (lines 9-11).

```
1  public class AnonymousPointcutASTVisitor extends BadSmellsASTVisitor {
2     private boolean visitAdvice(AdviceDeclaration node) {
3        isAnAnonymous(node.getPointcut());
4        return false;
5     }
6     protected void isAnAnonymous(PointcutDesignator pd) {
7        if (pd instanceof DefaultPointcut)
8           if (isAnAnonymousPointcut(((DefaultPointcut)pd).getDetail())){
9              BadSmellsEvent event = new BadSmellsEvent();
10             event.setType("Anonymous Pointcut Definition");
11             ...
12          }
13       ...
14    }
15 }
```

Listing 2. An AST visitor that detects the anonymous pointcut bad smell

2.2. Detecting Large Aspects

Whenever an aspect tries to deal with more than one concern, it could be divided in as many aspects as there are concerns. This smell is usually discovered when the developer finds several unrelated aspect members (fields, pointcuts, inter-type declarations) in the same aspect.

Definition 02 *Consider an aspect α. The crosscutting members of α are the collection of all advice, pointcuts, declare constructions and inter type declarations directly defined in α. Consider η as the number of crosscutting members of α. Given a threshold τ, an aspect is considered a large one whenever the predicate $\eta >= \tau$ holds. The function to determinate if an aspect is a large one, could be defined as: $f(\alpha) = \eta >= \tau$*

The threshold could be defined by the user of the function, or given as a constant. The detection in *AspectJ* could be implemented as a visitor (see Listing 3). All *TypeDeclaration* nodes are inspected in the end of the visiting process (line 2). Whenever the node

is an aspect, the number of declared members is obtained and compared to the τ value, defined in a constant named *TAU* available in a class named *Consts* (lines 4-5). If the number of crosscutting members is equal or higher than *TAU*, the aspect is marked as a *large aspect*, false otherwise.

```
1  public class LargeAspectASTVisitor extends BadSmellsASTVisitor {
2     public void endVisit(TypeDeclaration node) {
3        super.endVisit(node);
4        if (((AjTypeDeclaration) node).isAspect())
5           if (getNumberOfMembers() >= Consts.TAU){
6              BadSmellsEvent event = new BadSmellsEvent();
7              event.setType("Large Aspect");
8              ...
9           }
10    }
11 }
```

Listing 3. AST visitor responsible for the detection of the *Large Aspect* bad smell

2.3. Detecting Lazy Aspects

This *bad smell*, initially defined in [M.P. Monteiro 2005], occurs if an aspect has few responsibilities, and its elimination could result in benefits at the maintenance phase. Sometimes, this responsibility reduction is related to previous refactoring or to unexpected changes in requirements (planned changes that did not occur, for instance).

Definition 03 *Consider an aspect α. The crosscutting members of α are the collection of all advice, pointcuts, declare constructions and inter type declarations directly defined in α. Consider η as the number of crosscutting members of α. An aspect is considered a lazy one whenever the predicate $\eta == 0$ holds. The function could be defined as:* $f(\alpha) = \eta == 0$

To detect lazy aspects, a similar approach to the *Large Aspect* bad smell is taken. The *LazyAspectASTVisitor* creates bad smell events whenever an aspect without crosscutting members is found (see Listing 4).

```
1  public class LazyAspectASTVisitor extends BadSmellsASTVisitor {
2     public void endVisit(TypeDeclaration node) {
3        super.endVisit(node);
4        if (((AjTypeDeclaration) node).isAspect())
5           if (getNumberOfMembers() == 0){
6              BadSmellsEvent event = new BadSmellsEvent();
7              event.setType("Lazy Aspect");
8              ...
9           }
10    }
11 }
```

Listing 4. AST visitor responsible for the detection of the *Lazy Aspect* bad smell

2.4. Detecting Feature Envy

In *AspectJ*, pointcuts could be defined in aspects and also in classes. If a single aspect uses a class-defined pointcut, it is interesting to move the pointcut from the class to the aspect that uses it. The same problem might occur also in classes. It happens when a class extensively refers to members of another class instead of referring to its own. In this paper, we deal only with the detection of pointcuts in classes.

Definition 04 *If the number of pointcuts in a class χ is given by η, the class suffers from the feature envy bad smell if the predicate $\eta > 0$ holds. The function could be defined as:* $f(\chi) = \eta > 0$

The implementation using *AspectJ* is pretty straightforward (see Listing 5). The program checks all nodes representing types (aspects, classes and interfaces) and verifies if a class does not implement a pointcut in its body. If this happens, an event is generated. Note that the *visit(PointcutDeclaration node)* method (line 5) is executed only if the method *visit(TypeDeclaration node)* (line 2) returns true.

```
1  public class FeatureEnvyASTVisitor extends BadSmellsASTVisitor {
2     public boolean visit(TypeDeclaration node) {
3        return (!((AjTypeDeclaration) node).isAspect() || node.isInterface());
4     }
5     public boolean visit(PointcutDeclaration node){
6        BadSmellsEvent event = new BadSmellsEvent();
7        event.setType("Feature Envy");
8        ...
9        return false;
10    }
11 }
```

Listing 5. AST visitor responsible for the detection of the *Feature Envy* bad smell

2.5. Detecting Abstract Method Introductions

Aspects could be used to add state and behavior into existing classes. This is made through the inter-type declaration mechanism. This mechanism allows methods and/or fields to be inserted in classes defined by the aspect. However, the use of this functionality may cause problems when abstract methods are inserted in application classes.

This introduction forces the developer to provide concrete implementations to the introduced methods in every affected classes and sub-classes. This dependency unnecessarily increases the coupling between the aspect and the affected classes.

The introduction of abstract methods through an inter-type declaration should be avoided, since it demands providing implementation for these methods every time a subclass of the affected class is created.

Definition 05 *If the set of modifiers of a inter type method declaration ι is given by $m(\iota)$ and the abstract modifier is given by α, the function that describes if an inter type method declaration is an abstract one could be defined as $f(\iota) = \alpha \in m(\iota)$. If the result of the function evaluation is true, then the inter type declaration is an abstract one, false otherwise.*

An algorithm that detects this kind of smell could be seen in Listing 6. A *visit* method is defined to visit all inter type method declarations (line 2). If the *node* has *abstract* modifier, the inter type declaration is abstract, false otherwise (line 4).

```
1  public class AbstractMethodIntroductionASTVisitor extends BadSmellsASTVisitor {
2     public boolean visit(InterTypeMethodDeclaration node){
3        String name = node.getName().toString();
4        if (Modifier.isAbstract(node.getModifiers())){
5           BadSmellsEvent event = new BadSmellsEvent();
6           event.setType("Abstract Method Introduction");
```

```
7               ...
8           }
9           ...
10      }
11  }
```

Listing 6. An AST visitor that detects if an inter type method declaration is abstract

2.6. A Bad Smell Detector

A bad smell detector was implemented to test the algorithms described in this paper. It explores the AST support[3] available in the AJDT[4] project and is implemented as an *Eclipse* plugin.

This plugin extends both the *Eclipse* environment and the *AspectJ* environment. The *AspectJ* extension (Figure 1) was developed to provide mechanisms to find the bad smells discussed in this paper. The *Eclipse* extensions are available to provide visual information about the detected smells.

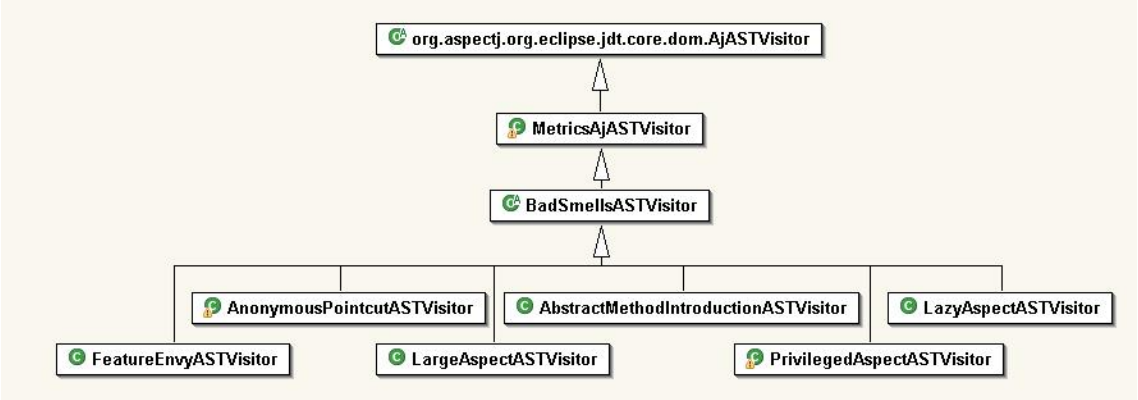


Figure 1. Class hierarchy of the AspectJ extension

The classes of the *AspectJ* extension package are briefly described here:

AjASTVisitor This class implements a visitor for abstract syntax trees. For each different concrete AST node type T there are some methods that could be used, such as *visit(T node)* or *endVisit(T node)*, to visit a given node and perform some arbitrary operation. This class is provided by the *AspectJ* reference implementation.

MetricsAjASTVisitor This class collects meta-information about the visited AST. It holds data about advice, pointcuts, inter type declaration fields, inter type declaration methods, declare constructions and size related metrics.

BadSmellsASTVisitor This visitor is responsible for reading information from eclipse files and collecting data to be displayed in the user interface. It is the direct superclass of all the bad smells AST visitors.

Other Classes There are other visitors defined to each type of bad smell being detected. Examples are: *AnonymousPointcutASTVisitor*, *AbstractMethodIntroductionASTVisitor*, *LazyAspectASTVisitor*, *FeatureEnvyASTVisitor*, *LargeAspectASTVisitor* and *PrivilegedAspectASTVisitor*[5].

[3] The developments in the AST support are still in progress and they are covered by enhancement `https://bugs.eclipse.org/bugs/show_bug.cgi?id=110465`.

[4] `http://www.eclipse.org/ajdt`

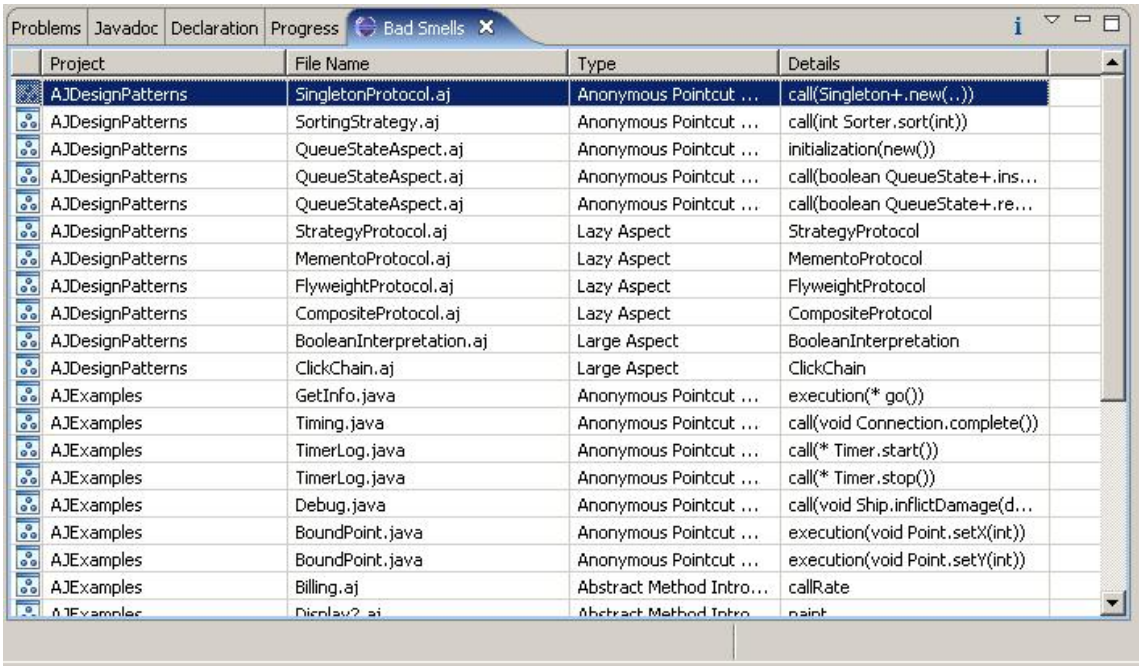
[5] This bad smell is not discussed in this paper

3. Case Study

This case study uses well-known aspect oriented programs available as open source. The selected systems give different flavors of *AspectJ* programs as they include tutorial examples, academic software, open source software and commercial application of the language.

The first system selected is the collection of examples shipped with the *AspectJ* language reference implementation (IBM). These examples aim to show the usage of the different constructions available to the language user. The second system is a collection of the GoF design patterns [Gamma et al. 1995] implemented using *AspectJ*. This collection was developed by Hannemman and Kiczales [Hannemann and Kiczales 2002] and it is used in other research papers [Garcia et al. 2005, M.P. Monteiro 2005]. The third system is a commercial product, developed by *GlassBox Corporation* and available as Open Source at Java.Net[6]. The *GlassBox Inspector* aims to deliver performance monitoring and troubleshoot mechanisms for *J2EE* applications using *AspectJ* and *JMX*.

In Figure 2, a view showing some occurrences of bad smells in the selected projects is shown. This view is populated whenever the user requires the activation of the detection plugin. For each bad smell, the following information is provided: project name, file name, type of bad smell and additional details.



Project	File Name	Type	Details
AJDesignPatterns	SingletonProtocol.aj	Anonymous Pointcut ...	call(Singleton+.new(..))
AJDesignPatterns	SortingStrategy.aj	Anonymous Pointcut ...	call(int Sorter.sort(int))
AJDesignPatterns	QueueStateAspect.aj	Anonymous Pointcut ...	initialization(new())
AJDesignPatterns	QueueStateAspect.aj	Anonymous Pointcut ...	call(boolean QueueState+.ins...
AJDesignPatterns	QueueStateAspect.aj	Anonymous Pointcut ...	call(boolean QueueState+.re...
AJDesignPatterns	StrategyProtocol.aj	Lazy Aspect	StrategyProtocol
AJDesignPatterns	MementoProtocol.aj	Lazy Aspect	MementoProtocol
AJDesignPatterns	FlyweightProtocol.aj	Lazy Aspect	FlyweightProtocol
AJDesignPatterns	CompositeProtocol.aj	Lazy Aspect	CompositeProtocol
AJDesignPatterns	BooleanInterpretation.aj	Large Aspect	BooleanInterpretation
AJDesignPatterns	ClickChain.aj	Large Aspect	ClickChain
AJExamples	GetInfo.java	Anonymous Pointcut ...	execution(* go())
AJExamples	Timing.java	Anonymous Pointcut ...	call(void Connection.complete())
AJExamples	TimerLog.java	Anonymous Pointcut ...	call(* Timer.start())
AJExamples	TimerLog.java	Anonymous Pointcut ...	call(* Timer.stop())
AJExamples	Debug.java	Anonymous Pointcut ...	call(void Ship.inflictDamage(d...
AJExamples	BoundPoint.java	Anonymous Pointcut ...	execution(void Point.setX(int))
AJExamples	BoundPoint.java	Anonymous Pointcut ...	execution(void Point.setY(int))
AJExamples	Billing.aj	Abstract Method Intro...	callRate
AJExamples	Display2.aj	Abstract Method Intro...	paint

Figure 2. A view showing the bad smells in the case study

In an attempt to define appropriated thresholds for the *Large Aspect* bad smell, the examples chosen (plus AJHotDraw [van Deursen et al. 2005]) were measured regarding the number of crosscutting members of its aspects. The data was analyzed and the negative binomial statistical distribution with the following parameters was selected: $NegBin(3, 0.43034)$. This distribution was selected from the results of the Chi-Square test application in the input data.

[6] `https://glassbox-inspector.dev.java.net/`

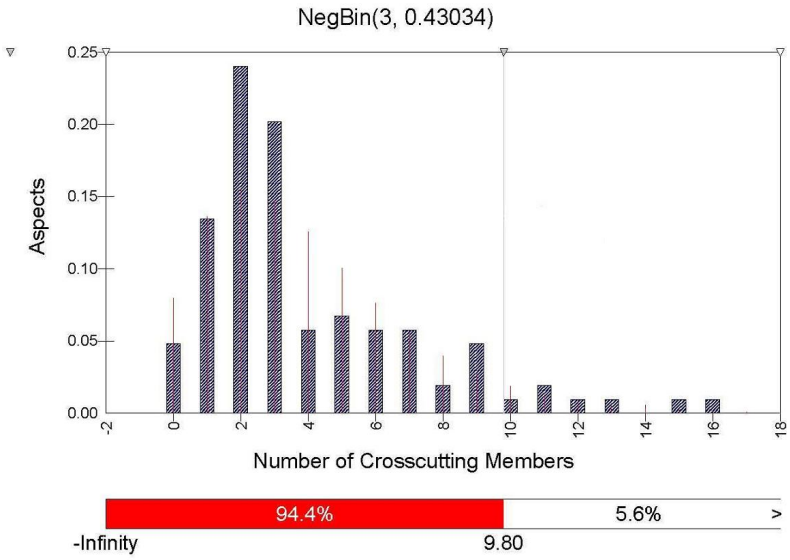


Figure 3. The binomial distribution used to represent the number of crosscutting members in aspects

Using this theoretical distribution, representing a subset of the existing aspect oriented systems, one can choose a coherent τ value. In this case, 94.4% of the aspects have less than ten crosscutting members. The Figure 3 shows the relation between *aspects* and *crosscutting members*. The dark bars represent the number of crosscutting members per percentage of aspect. The red lines represent the negative binomial distribution used. In this case study, aspects with ten or more crosscutting members are marked as *large aspects*.

In next sections, each system is detailed as following: first, a brief description about the system under evaluation is presented. After that, a table summarizing the detected bad smells is presented and each type of bad smell is discussed regarding its instances in the application.

3.1. System 1: AspectJ Examples

The *AspectJ* examples provide illustrative source code to teach the users on the development of aspect-oriented programs using the language. These examples are divided into categories, such as: development aspects, tracing using aspects, production aspects and reusable aspects.

Each example works with different facets of the language. The domains used in those examples vary from telecom simulation and space war game to tracing systems. There is also an implementation of a reusable Observer pattern [Gamma et al. 1995] as an example.

In Table 1, the occurrences of each bad smell are summarized. The *anonymous pointcut definition* bad smell is the one that appears most (7 cases). No instance of a *lazy aspect* was found and a *large aspect* was detected in one of the examples. *Feature envy* and *abstract method introduction* appear in a few aspects.

Type	Number of Occurrences
Classes	46
Aspects	27
Interfaces	5
Bad Smell	
Anonymous Pointcut Definition	22 of 52 advice
Large Aspect	1 of 27 aspects
Lazy Aspect	0 of 27 aspects
Feature Envy	1 of 46 classes
Abstract Method Introduction	3 of 28 inter-type methods

Table 1. Bad Smells in AspectJ Examples

As an example of the *anonymous pointcut definition* bad smell, the pointcut *demoExecs() && !execution(* go()) && goCut()* declared in an aspect named *GetInfo* is composed by two defined pointcuts (*demoExecs* and *goCut*) and an anonymous pointcut definition (*!execution(* go())*). This last piece could be extracted in a new pointcut and its name used instead of the literal predicate. The resulting composition would be, for example: *demoExecs() && !goExecs() && goCut()*. Other detected occurrences of this bad smell could be found in the *Timing*, *TimerLog*, *Debug* and *BoundPoint* aspects.

The high number of occurrences of this specific bad smell is due to the nature of the examples. Each example is intended to cover specific features of the language, without taking reuse concerns in all applications. While good design techniques are desired, some of them may introduce unnecessary complexity to those that are trying to learn a new language (the main audience of the examples).

The aspect detected as a *large aspect* is the *Debug* aspect. It defines advice dealing with different concerns simultaneously. This aspect collects points regarding user interface modification, changes in the registry contents, and ship collisions, among other concerns. Although all of these features are related to system debugging, they could be divided in several aspects, each one with a different perspective on debugging. The opposite (*lazy aspects*) were not found in the examples.

Feature envy is present in the *Ship* class, which implements a spaceship in the *SpaceWar* example. This class contains a pointcut definition that is used only in the *EnsureShipIsAlive* aspect. The coupling between class and aspect is reduced, and the aspect's cohesion is improved if the pointcut definition moves to the aspect.

An *abstract method introduction* exists in the *Billing* aspect, which charges for telephone calls according to the type and length of a performed call. So, the user of the class that receives the introduction should be aware of which aspects affect the code, and then, add methods to the aspect. This dependency could increase the solution's complexity.

3.2. System 2: AspectJ Design Patterns

Hanneman and Kiczales [Hannemann and Kiczales 2002] describe an experiment where the *gang of four* (GoF) design patterns [Gamma et al. 1995] were implemented in both *Java* and *AspectJ*. The authors state that aspect-oriented implementations have improved modularity in 17 of the 23 studied cases.

The degree in which the enhancement occurs depends on the relationship among

the roles played by the classes and objects within the pattern. Those patterns where an object plays more than one role, or where several objects play the same role, had the most significant improvement.

Garcia et al [Garcia et al. 2005] performed measurements on implementations of the GoF design patterns using quality metrics referring to separation of concerns, coupling, cohesion, and code size. The authors state that, in several cases, the aspect-oriented solution improved the separation of concerns relative to the participating roles of the design patterns.

Table 2 shows the occurrences of each type of bad smell. The *anonymous pointcut definition* bad smell appears in five situations. *Lazy aspects* were found four times and two *large aspects* were detected in the patterns. *Feature envy* and *abstract method introduction* do not appear in these examples.

Type	**Number of Occurrences**
Classes	88
Aspects	42
Interfaces	16
Bad Smell	
Anonymous Pointcut Definition	5 of 15 advice
Large Aspect	2 of 42 aspects
Lazy Aspect	4 of 42 aspects
Feature Envy	0 of 88 classes
Abstract Method Introduction	0 of 39 inter-type methods

Table 2. Bad Smells in AspectJ Design Patterns

A first occurrence of the anonymous pointcut bad smell occurs in the *SingletonProtocol* aspect: *call((Singleton+).new(..)) && !protectionExclusions()*. Instead, a composed pointcut could be used (*singletonCreation() && !protectionExclusions()*).

The second occurrence belongs to an aspect named *SortingStrategy*. The predicate contains a *call* primitive: *call(int[] Sorter.sort(int[]))*. This predicate affects only the calls to the *Sorter.sort* method. It appears in an around advice. The advice code could be inserted directly in the *sort* method. The same happens with the pointcut *initialization(new()) && target(queue)* in the *QueueStateAspect*. The code triggered by the advice could be inlined in the constructor. Other examples of this smell could be found in the *QueueStateAspect* aspect.

The *lazy aspect* bad smell appears in four aspects: *StrategyProtocol*, *MementoProtocol*, *FlyweightProtocol* and *CompositeProtocol*. These aspects do not have any crosscutting members and could be safely converted to classes. Whenever an aspect does not have members implementing crosscutting concerns a class could (and should, if possible) be used instead.

The first *large aspect* is the *BooleanInterpretation* aspect. It is responsible for adding methods to perform the *replace* and *copy* operations in the following classes: *AndExpression*, *BooleanConstant*, *OrExpression*, *VariableExpression*, *NotExpression*. To provide those methods, ten inter type method declarations were used. The aspect could be broken in two aspects (one for the copy additions, another for the replace operations) or into five separated aspects: one for each affected class.

The second *large aspect* (named *ClickChain*) uses four parent declarations

(*Frame*, *Panel* and *Button* implements *Handler* and *Click* implements *Request*) and defines inter type declaration methods to add *handle* and *accept* behavior to the *Button*, *Panel* and *Frame* classes. It also defines a pointcuts to handle clicks in the *ChainOfResponsibility* pattern implementation. This aspect could be divided per affected classes (one aspect for affected class) or per operation (*handle* or *accept*).

Occurrences of the *feature envy* and *abstract method introduction* were not detected in the examples.

3.3. System 3: Glassbox Inspector

The *Glassbox Inspector* project uses *AspectJ* and *JMX* to monitor performance for Java/J2EE applications. It provides information to identify specific problems, capture statistics, monitor database calls etc. The version used in this case study was version 1.0 beta.

Table 3 summarizes the occurrences of bad smells in the *Glassbox*. The *anonymous pointcut definition* bad smell appears in seven places in the system. Two *large aspects* and one *lazy aspect* are present in the source code. *Feature envy* and *abstract method introduction* do not appear in these examples.

Type	**Number of Occurrences**
Classes	12
Aspects	26
Interfaces	7
Bad Smell	
Anonymous Pointcut Definition	7 of 27 advice
Large Aspect	2 of 26 aspects
Lazy Aspect	1 of 26 aspects
Feature Envy	0 of 12 classes
Abstract Method Introduction	0 of 17 inter-type methods

Table 3. Bad Smells in GlassBox

The first three anonymous pointcuts appear in the *TraceJdbc* aspect. The predicate *call(* java.sql..*(..))* || *call(* javax.sql..*(..))* is the same in all advice. The predicate could be extracted in a single pointcut definition and the name of the new pointcut used in the pieces of advice. Other occurrences of the same bad smell could be found in the *LogManagement*, *AbstractOperationMonitor* and *AbstractRequestMonitor* aspects.

Two aspects were detected as large ones. The aspect named *LogManagement* has thirteen crosscutting members. Eight of them are inter type method declarations that provide basic functionality for classes that should be logged. Instead of having methods such as: *logError(...)*, *logWarn(...)*, *logInfo(...)* and *logDebug(...)*, the developers could replace them by a general solution, passing the severity as a formal argument: *log(..., Severity severity)*.

The *ErrorHandling* aspect has eleven crosscutting members but does not need to be reduced. There is a pointcut named *handlingScope* that composes five other pointcuts and is used by an around advice. This advice ensures that errors in the monitoring code will not damage the underlying application code. As the pointcut predicate is a large one, the developers split the predicate into five others.

One *lazy aspect* was detected in the Glassbox Inspector. The *AbstractResource-*

Monitor aspect does not have crosscutting members, but it could not be converted to a class because it extends the *AbstractRequestMonitor* aspect (in *AspectJ*, classes could not extend aspects).

Feature envy and *abstract method introduction* were not detected in the *Glassbox Inspector*.

4. Related Work

The method described by Simon et al [Simon et al. 2001] uses metrics to detect bad smells. In particular, the method tries to detect candidates for the following refactorings: *Move Method*, *Move Attribute*, *Extract Class* and *Inline Class*. An equation is presented to evaluate the cohesion of methods and attributes inside the classes of a system. The results are converted to a three-dimensional Cartesian coordinate system, and then rendered visually. Similar approaches are used in [Lanza and Ducasse 2002] and [van Emden and Moonen 2002].

The Daikon tool presented in [Kataoka et al. 2001] uses program invariant detection to find suitable applications of refactorings. Invariants are values that remain constant every time some piece of code is executed, and indicate the possible application of refactorings. The detection process implicates in the instrumentation of the code for analysis during runtime, and the execution of a comprehensive set of tests, so the tool can analyze a wide range of possible interactions.

Tourwé and Mens [Tourw and Mens 2003] propose the use of logic metaprogramming. Logic programming statements are used to detect bad smells such as *obsolete parameters* and *inappropriate interfaces*.

The tactics of [Balazinska et al. 2000] and [Ducasse et al. 1999] are similar in that both attempt to find repeated sections of source code throughout a software system. The former approach focuses on Java code – and thus involves the parsing of the code, while the latter tries to remain language independent, considering the source code only as text strings. As a result of this, the first, while more limited in scope, can make use of information (such as the context in which a method is used) to provide a more precise analysis. A few other approaches to automate the detection of bad smells in software systems are presented in [Mens and Tourwe 2004].

5. Conclusion

In this paper we discussed some algorithms to detect bad smells in *AspectJ* programs. The defined algorithms could be extended to deal with more special cases of each type of bad smell. Other algorithms could also be defined for different types of bad smells, such as *privileged aspects*, *code duplication* and *inappropriate intimacy*.

The provided implementation could be extended to support those other kinds of smells. Additional systems may be subject of further investigation. The appropriate detection and removal of bad smells could affect quality attributes in the software being modified and each refactoring might be evaluated regarding those attributes.

The evaluated systems in the case study have, in general, a low number of smells. The one that appears more frequently is the *anonymous pointcut definition*. This smell

is usually removed whenever the predicate is used in more than one advice/inter-type declaration or when the aspect is an abstract one.

Some smells such as *feature envy* and *abstract method introduction* are less frequently detected. The *large aspect* detection depends on the definition of significant thresholds. This definition could be gathered from the analysis of existing systems or provided by the users of the detection tool. *Lazy aspect* occurrences are associated with aspects that do not have crosscutting members and could be replaced by classes without problems.

6. Acknowledgments

We would like to thank: the AspectJ team and Andrew Huff for providing the initial AST support in AspectJ, Andy Clement and Helen Hawkins for their constant and quick feedback about the AST implementation, Rafael Chaves for providing useful directions on Eclipse plugin development and Deise Saccol for her comments on drafts of this paper. This work has been partially supported by CNPq under grant No.140046/2006-2 for Eduardo Piveta.

References

[Balazinska et al. 2000] Balazinska, M., Merlo, E., Dagenais, M., Lage, B., and Kontogiannis, K. (2000). Advanced clone-analysis to support object-oriented system refactoring. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 98, Washington, DC, USA. IEEE Computer Society.

[Boehm and Sullivan 2000] Boehm, B. W. and Sullivan, K. J. (2000). Software economics: a roadmap. In *ICSE - Future of SE Track*, pages 319–343.

[Ducasse et al. 1999] Ducasse, S., Rieger, M., and Demeyer, S. (1999). A language independent approach for detecting duplicated code.

[Elssamadisy and Schalliol 2002] Elssamadisy, A. and Schalliol, G. (2002). Recognizing and responding to bad smells in extreme programming. In *Proceedings of the 24th International conference on Software Engineering*.

[Fowler et al. 2000] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (2000). *Refactoring: improving the design of existing code*. Object Technology Series. Addison-Wesley.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. Addison-Wesley.

[Garcia et al. 2005] Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., and von Staa, A. (2005). Modularizing design patterns with aspects: A quantitative study. In *4th International Conference on Aspect-Oriented Software Development (AOSD'05)*.

[Garcia et al. 2004] Garcia, V. C., Piveta, E. K., Lucrédio, D., Alvaro, A., de Almeida, E. S., do Prado, A. F., and Zancanella, L. C. (2004). Manipulating crosscutting concerns. *4th Latin American Conference on Patterns Languages of Programming (SugarLoafPlop 2004)*.

[Hanenberg et al. 2003] Hanenberg, S., Oberschulte, C., and Unland, R. (2003). Refactoring of aspect-oriented software. In *Net.Object Days 2003*.

[Hannemann and Kiczales 2002] Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press.

[Iwamoto and Zhao 2003] Iwamoto, M. and Zhao, J. (2003). Refactoring aspect-oriented programs. In *The 4th AOSD Modeling With UML Workshop, UML'2003*.

[Kataoka et al. 2001] Kataoka, Y., Ernst, M., Griswold, W., and Notkin, D. (2001). Automated support for program refactoring using invariants.

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In Knudsen, J. L., editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin. Springer-Verlag.

[Kiczales et al. 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In Akşit, M. and Matsuoka, S., editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag.

[Lanza and Ducasse 2002] Lanza, M. and Ducasse, S. (2002). Understanding software evolution using a combination of software visualization and software metrics.

[Mens and Tourwe 2004] Mens, T. and Tourwe, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

[Monteiro and Fernandes 2004] Monteiro, M. P. and Fernandes, J. M. (2004). Object-to-aspect refactorings for feature extraction. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD'2004)*. ACM Press.

[M.P. Monteiro 2005] M.P. Monteiro, J. F. (2005). Towards a catalog of aspect-oriented refactorings. In Mezini, M., editor, *Proc. 4th, Int Conf. on Aspect-Oriented Software Development (AOSD-2005)*. ACM Press.

[Piveta et al. 2005] Piveta, E., Hecht, M., Pimenta, M., and Price, R. T. (2005). Bad smells em sistemas orientados a aspectos (in portuguese). *Brazilian Symposium on Software Engineering, SBES 2005, Uberlandia - Brasil*.

[Simon et al. 2001] Simon, F., Steinbruckner, F., and Lewerentz, C. (2001). Metrics based refactoring. In *CSMR*, pages 30–38.

[Tourw and Mens 2003] Tourw, T. and Mens (2003). Identifying refactoring opportunities using logic meta programming.

[van Deursen et al. 2005] van Deursen, A., Marin, M., and Moonen, L. (2005). Ajhotdraw a showcase for refactoring to aspects. In *Linking Aspect Technology and Evolution (AOSD-2005)*.

[van Emden and Moonen 2002] van Emden, E. and Moonen, L. (2002). Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press.