3
Code Anomalies in Aspect-Oriented Programming

As pointed out in the first chapter, this research work is going to tackle the issue of understanding architecture degradation symptoms through the analysis of code anomalies. The goal is to perform this investigation taking into consideration software systems structured with different modularization techniques such as: object-oriented programming and aspect-oriented programming (Kiczales *et al.*, 1997). These techniques have been selected due to existing initial evidence about how the inappropriate modularization of architectural concerns impact the architectural design.

However, the analysis of code anomalies in aspect-oriented systems is particularly hindered by the fact that existing catalogs of anomalies in such systems are very limited (Section 2.3.4.1). In addition, there is no knowledge about whether and how often the documented aspect-oriented anomalies manifest themselves in system implementations (Section 2.3.4.1). Consequently, before analyzing the impact of code anomalies in architecture degradation symptoms, including those anomalies that infect object-oriented and aspect-oriented implementations, we need establish catalogs of anomalies that represent recurrent misuses of aspect-oriented mechanisms.

In this context, this chapter presents the characterization and classification of six (06) new code anomalies frequently introduced by developers in aspect-oriented programs (Section 3.2.2). These anomalies are associated with various mechanisms of aspect-oriented programming (Section 3.1). We illustrate concrete examples of how the proposed anomalies manifest in system implementation. Additionally, for each proposed anomaly we define a detection strategy (Section 2.3.1) that supports the anomaly identification. In order to further evaluate to what extent the already documented (Section 3.2.1) and the new anomalies occur in wider contexts, an empirical study was carried out involving three software systems (Section 3.3). In addition, the study assesses which code anomalies are likely to manifest themselves more often in aspect-oriented systems. It is

important to note that this study is not concerned about discussing how aspect-oriented anomalies might impact on a system architecture decomposition, this will be discussed in Chapter 4. Finally, the chapter summarizes the key points in Section 3.4. All the information presented in this chapter has been reported in four technical papers (Macia, 2011; Macia *et al.*, 2010; Macia *et al.*, 2011a; Macia *et al.*, 2011b).

3.1. Aspect-Oriented Programming

This section presents key concepts and mechanisms of the aspect-oriented programming. The understanding of such concepts and mechanisms is fundamental for the comprehension of the aspect-oriented anomalies introduced later in this chapter.

Aspect-Oriented Programming (AOP) (Kiczales *et al.*, 1997) is a modularization technique that supports a new flavor of separation of concerns that crosscut multiple elements at the implementation level. To this end, AOP introduces a new modularization abstraction called *aspect* and new composition mechanisms called *pointcut* and *advice*.

Aspect is the term used to denote the abstraction that aims to support improved isolation of crosscutting concerns. Aspects are modular units of crosscutting concerns that are associated with a set of classes. An aspect can affect, or crosscut, one or more classes in different ways. In AOP, aspects modularize crosscutting concerns and classes modularize non-crosscutting concerns. Besides conventional attributes and methods, an aspect includes *pointcuts* and pieces of *advice* as described below.

Pointcut expression (or just pointcuts) is a first-order logic expression that selects the join points that will be affected by the aspect crosscutting behavior. *Join points* are well-defined points in the dynamic execution of a system. They specify how classes and aspects are related. Examples of join points are method calls, method executions, writing and reading of attributes, and object initialization.

Advice is a special method-like construct attached to pointcuts. The advice is executed when the program execution reaches a join point selected by some

pointcut expression. There are different types of advice: (i) a *before* advice runs whenever a join point is reached and before the actual computation proceeds, (ii) an *after* advice runs after the computation under the join point finishes, i.e. after the method body has run, and just before control is returned to the caller, and (iii) an *around* advice runs whenever a join point is reached, and has explicit control whether and when the computation under the join point is allowed to run if it is executed at all.

AspectJ (Kickzales *et al.*, 2001) is the most well-known and used language for AOP. It is an extension to the Java programming language. Besides the aforementioned concepts, aspects in AspectJ can provide intertype declarations that allow defining parent classes and insert attributes and methods into classes. Figure 3.1 illustrates how AOP mechanisms are supported by AspectJ using an example extracted from Health Watcher, a web software system (Soares *et al.*, 2002; Greenwood *et al.*, 2006) that is used in our case studies. The HWDataCollection consists of an inter-type declaration (lines 03 and 04), a pointcut (line 06), an advice (lines 08-17). This covers the basics of what aspects can contain.

The inter-type declaration defines that SystemRecord is the superclass of ComplaintRecord, SpecialityRecord, HealthUnitRecord, and EmployeeRecord. The pointcut named record defines as join points the call to constructor of any descendant of SystemRecord and that call is not performed within the aspect HWDataCollection or any of its descendants. Finally, the advice creates a system record depending on the join point type.

```
01  public aspect HWDataCollection {
02    private interface SystemRecord {};
03    declare parents: ComplaintRecord || SpecialityRecord || HealthUnitRecord ||
04                     EmployeeRecord implements SystemRecord;
05
06    pointcut records: call(SystemRecord+.new(..)) && !within(HWDataCollection+);
07
08    Object around (): records() {
09      RepoFactory factory = RepoFactory.getRepositoryFactory();
10      Class type = getSystemRecord(thisJoinPoint.getSignature().getDeclaringType());
11      if (type.equals(ComplaintRecord.class)) {..}
12      else if (type.equals(HealthUnitRecord.class)) {..}
13      else if (type.equals(MedicalSpecialityRecord.class)) {..}
15      ...
16      return null;
17    }
18
19  }
```

Figure 3.1: Example of an aspect in AspectJ.

3.2. Identification of Code Anomalies

This section presents the set of aspect-oriented anomalies investigated through this research work, including both the cases of already published and new code anomalies. The investigated code anomalies were classified in three categories. The first category comprises anomalies related to *anomalous pointcut* definitions, such as duplication and complexity. The second group, *aspect definition*, is formed by anomalies resulting from inappropriate modularization of crosscutting concerns into aspects. The anomalies in the third category, *undesirable interdependencies*, occurring due to modularization anomalies involving two or more crosscutting concerns, thereby leading to tight dependencies: (1) between an aspect and the base code, or (2) between two or more aspects. We presented a brief description of documented anomalies and their key characteristics (Section 3.2.1), the newly-revealed code anomalies are presented with more emphasis on (Section 3.2.2).

3.2.1. A Catalog of Already Documented Code Anomalies

This research relies on eight already documented anomalies because they were described in a systematic fashion and, therefore, could be more precisely identified. For instance, aspect-oriented code anomalies described by short and abstract definitions were not considered. The reason for this is that such

definitions make code anomalies difficult to precisely interpret and also to define strategies for their detection (Iwamoto *et al.*, 2003; Hannemann *et al.*, 2005). It is important to highlight that we identified limitations (in terms of coverage of AOP mechanisms) in some of these code anomalies. Such limitations were addressed through the definition of a code anomaly catalog (Section 3.3).

Five anomalies are proposed in (Srivisut and Muenchaisri, 2007) while the others are defined in (Piveta *et al.*, 2006). The following 4 code anomalies are related *to pointcut declarations* and, therefore, they are classified in the first category. *Duplicate Pointcut (DP)* (Srivisut and Muenchaisri, 2007) occurs whenever different pointcut definitions collect the same set of join points in base code, and *Anonymous Pointcut (AP)* (Piveta *et al.*, 2006) occurs whenever a pointcut is directly defined in the advice signature. *Junk Material (JM)* (Srivisut and Muenchaisri, 2007) refers to cases of pointcuts not referred by any advice. *Borrowed Pointcut (BP)* (Srivisut and Muenchaisri, 2007) is a pointcut that is referred by other aspects, which are not subaspects of the one in which the pointcut is actually defined. We have considered the original definition of already published anomalies, such as the case of *Borrowed Pointcuts*, independently if we agreed or not with the characterization of the code structure as an anomaly.

The following three anomalies are related to *aspect definitions* and, as a consequence, they are classified in the second category. *Lazy Aspect (LA)* (Piveta *et al.*, 2006) is an aspect that has either none or only fragmented responsibility. An aspect is marked as realizing *Various Concerns (VC)* (Srivisut and Muenchaisri, 2007) when the pointcut is associated with more than one advice addressing different concerns. *Abstract Method Introduction (AMI)* (Srivisut and Muenchaisri, 2007) occurs whenever an abstract method is added into an existing class using inter-type declaration. Finally, *Feature Envy (FE)* (Piveta *et al.*, 2006) falls in the category of *undesirable dependencies* and occurs whenever there are pointcuts defined within a class, and these pointcuts are externally referred by aspects.

3.2.2.
A Catalog of New Code Anomalies

The discovery of the new code anomalies was mainly driven by our experience in the development of aspect-oriented systems, our observations and analysis of potentially-anomalous code structures and maintenance effort in several systems: Telecom (2009), a Design Pattern library (Hannemann and Kiczales, 2002), and Health Watcher (Soares *et al.*, 2002; Greenwood *et al.*, 2007). In the proposed catalog, each anomaly is defined by a textual description and potential relationships with other anomalies, a concrete example and a detection strategy (Marinescu, 2004; Lanza and Marinescu, 2006). The discussed examples were extracted from the software systems used in our case study (Section 3.3).

As far as the detection strategies are concerned, they are structured in the form *name<entity> := condition*. The *name* corresponds to the anomaly name that the strategy detects. The *entity* indicates the code element type (i.e. aspect, pointcut or advice) over which the strategy is applied. The *condition* part compasses the combination of one or more measures outcome related to the element under analysis. The definition of the strategies also relies on symbolic constants in the place of thresholds (e.g. LOW and HIGH). The choice of these values will depend on the system characteristics and programmers styles.

Furthermore, the detection strategies were defined on the basis of previous guidelines reported in the literature (Marinescu, 2004; Lanza and Marinescu, 2006). In order to define the detection strategies for aspect-oriented anomalies, we relied on available metrics for aspect-oriented programs (Sant'Anna *et al.*, 2007; Srivisut and Muenchaisri, 2007), which were useful to identify modularity anomalies in previous studies (Soares *et al.*, 2002; Greenwood *et al.*, 2007; Sant'Anna *et al.*, 2007; Figueiredo *et al.*, 2009).

3.2.2.1.
Anomalous Pointcut Definition

The code anomalies classified in the category of anomalous pointcut declarations are: *God Pointcut*, *Redundant Pointcut*, and *Idle Pointcut*. Each anomaly is presented in terms of a description, an example and, a detection

strategy. As aforementioned, detection strategies are organized following Marinescu's guideline (Marinescu, 2004; Lanza and Marinescu, 2006).

3.2.2.2. God Pointcut

Definition. *God Pointcut (GP)* occurs when: (i) a pointcut has either a complex expression involving many keywords *or* picks out many scattered join points, and (i) the respective advice has a complex implementation. When this occurs, the pointcut expressions and advice are very difficult to read. As a consequence, their decomposition would improve the readability and increase their chance to be reused.

Example. An example of *God Pointcut* can be observed in the pointcut synchronizationPoints in the iBATIS system, Figure 3.2. This pointcut is interested in picking out the execution of specific methods. However, its definition specifies, without using wildcards, more than 10 method executions and more than 12 logic operators. In addition, its advice has more than 30 lines of code.

```
public aspect IBatisSynchronizationExecutingObject extends PessimisticSynchronization{
  ...
  protected pointcut synchronizationPoints(Object syncObj):
      ((execution(public void RuntimeException.setNextException(Exception)) ||
        execution(protected static int BaseLogProxy.getNextId()) ||
        execution(public void FifoController.putObject(CacheModel,Object,Object)) ||
        ...
        execution(public void LruController.putObject(CacheModel,Object,Object)) ||
        ...
        execution(public void MemoryController.putObject(CacheModel,Object,Object)) ||
        ...
        execution(private void LazyResultLoader.loadObject() throws RuntimeException) ||
        execution(public Object[] ResultMap.getResults(Request,Set) throws SQLException) ||
        execution(public static long SessionScope.getNextId()) ||
        execution(public Object OSCacheController.removeObject(CacheModel,Object)) ||
        execution(private void CacheModel.getObjectPartTwo())) && this(syncObj));
  ...
}
```

Figure 3.2: Example of *God Pointcut*.

Detection Strategy. In order to detect this code anomaly we defined the detection strategy presented below. We always refrained from not implementing overly complex detection strategies, i.e. involving many metrics and threshold values. This policy enabled us to have a better control of threshold variability and adjustments, and more control of potential false negatives (Chapter 4). As a pointcut can be associated with one or more advice, we considered it as a *God Pointcut* when the following detection strategy is true for at least one of its advice:

GP<pointcut> := (SPP > HIGH **or** SJP > HIGH) **and**
(CC > HIGH **or** LOC > VERY_HIGH)

The metric SPP (*Set of Primitive Pointcuts referring a pointcut*) counts the number of primitive pointcuts (e.g. call, execution) used in the pointcut expression. The metric SJP (*Set of Join Points*) computes the Set of Joinpoint shadows captured by a given *pointcut*. The metric CC (*Cyclomatic Complexity*) corresponds to the complexity of the advice that referring a given pointcut (McCabe, 1976). Finally, the metric LOC (*Line Of Code*) counts the number of lines of code involving in the advice implementation. We are aware that SJP metric could be a limited indicator because there are some primitive pointcuts of AspectJ that pick out join points based heavily on the particular executions of a system. This discussion is similar to the discussions on the effective use of static and dynamic metrics Arisholm (2002), which is another point to be investigated and is outside of the scope of this thesis.

3.2.2.3. Idle Pointcut

Description. *Idle Pointcut (IdP)* is associated with pointcuts, which do not match any join point. There are multiple causes for this problem. First, mistakes in the pointcut expression may cause that any join point is captured. Second, the pointcuts may no longer capture the intended join points as a consequence of refactorings on the base code. Finally, pointcuts are not referred to by any advice, and, therefore, no action is performed when the join points are reached. *Idle Pointcut* is a case of code anomaly that was created to address limitations in the definition of existing code anomalies. More specifically, it enables to capture anomalies that are not covered by *Junk Material* (Section 3.2.1), which is only focused on cases of pointcuts not referred by any advice.

Example. Figure 3.3 shows a typical scenario for this problem. The callSqlExecuteUpdate pointcut, highlighted in gray, is part of the aspect SqlmapEngineMappingECAspect implementation. The expression picks out calls to the sqlUpdate method. Such calls should be fired within executeUpdate or executeQueryCall methods from the GeneralStmt class. However, this pointcut was broken during the refactoring of the method executeUpdate. This method uses another method called executeUpdatePartOne, highlighted in gray in the base code. This refactoring resulted in an *Idle Pointcut* instance.

```
//base code
public class GeneralStatement extends BaseStatement{
  ...
  public int executeUpdate() throws SQLException {
     rows = executeUpdatePartOne(..);
  }
}

//aspect code
public aspect SqlmapEngineMappingECAspect{
  ...
  public pointcut callSqlExecuteUpdate(): call(protected int sqlUpdate(..))
            && withincode(public int GeneralStmt.executeUpdate(..)) ||
               withincode(protected List GeneralStmt.executeQueryCall(..)));
  ...
}
```

Figure 3.3: Example of *Idle Pointcut*.

Detection Strategy. Based on the aforementioned characteristics of the *Idle Pointcut* anomaly we defined the detection strategy as follows:

IdP<pointcut> := (SJP = 0) **or** (NAdP = 0)

The metric SJP (*Set of the corresponding Join points of a given pointcut*) counts the number of join points picked out by a given pointcut. The metric NAdP (*Number of Advice referring to a Pointcut*) (Srivisut and Muenchaisri, 2007) counts the number of pieces of advice that are related to a given pointcut.

3.2.2.4. Redundant Pointcut

Description. Pointcuts can be reused or combined by logical operators in order to define new composite pointcuts. The *Redundant Pointcut (RP)* code anomaly is associated with partial (not full) pointcut expressions equivalent to others that have already been defined. This code anomaly can be characterized as a variation of *Duplicate Pointcut* (Srivisut and Muenchaisri, 2007) because it considered only a subset of the expressions used in the pointcut definition.

Example. The example in Figure 3.4 depicts two advice defined in the context of the ComplaintStateAspect. These advice are interested in capturing initializations of classes whose names match the "*Complaint*" prefix and such initializations are made by a Complaint object. In this context, the advice duplicated the definition of this requirement rather than isolating it in a single pointcut expression.

```
public aspect ComplaintStateAspect{
  ...
  after(Complaint complaint): initialization(Complaint+.new()) && target(complaint){…}

  after(Complaint complaint, String complainer, String description,..):
            initialization(Complaint+.new()) && target(complaint) &&
            args(complainer, description, ..) {…}
  ...
}
```

Figure 3.4: Example of *Redundant Pointcut*.

Detection Strategy. Based on the aforementioned characteristics of the Redundant Pointcut anomaly its corresponding detection strategy is defined as follows:

RD<pointcut, advice> := (NPJP > 0)

The metric NPJP (*Number of Pointcuts with Join Points in common*) counts the number of pointcuts or advice that have two or more primitive expressions in common (i.e. that attempt the same of set of join points) with a given pointcut.

3.2.2.5. Undesirable Interdependencies

This subsection describes the three code anomalies in this category: *Forced Join* Point, *God Aspect* and *Composition Bloat*. Similarly to previous section, these anomalies are described in terms of a description, example and, a detection strategy.

3.2.2.6. Forced Join Point

Description. *Forced Join Point (FJP)* is associated with elements (attributes or methods) in the base code that are only exposed to be used by aspects. For instance, they might be methods in the base code whose implementation details are exposed in their signatures just for the sake of passing internal module information to the aspects. There are also instances of temporary class fields, which reveal information that otherwise would be associated with local variables. They might also represent cases of hook methods for enabling the capture of certain method calls. As a result, some possible side effects are that

these base methods end up either implementing a non-cohesive part of an algorithm or having empty bodies.

Example. Figure 3.5 presents an example of *Forced Join Point* extracted from Health Watcher system (2009). This example includes a method called init defined as part of the implementation, in the base code, of the class HWServlet. Note that the implementation of this method is very simple so its execution could be exposed as events to the aspects. Specifically, it is picked out by the advice defined in the aspect ServletCommanding. This advice is responsible for registering more than thirty command objects that will be used in the system.

```
// base code
public class HWServlet extends HttpServlet{
  ...
  public void init(..) throws ServletException{
    facade = HealthWatcherFacade.getInstance();
  }
}
//aspect code
public aspect ServletCommanding{
  ...
  after(): execution(void HWServlet.init(..)){
   commandTable = new Hashtable();
   registerCommand(CommandConfigRMI, new ConfigRMI());
   registerCommand(CommandGetDataByDiseaseType, new GetDataByDiseaseType());
   registerCommand(CommandGetDataByHealthUnit, new GetDataByHealthUnit());
   registerCommand(CommandGetDataSpeciality, new GetDataBySpeciality());
   registerCommand(CommandInsertAnimalComplaint, new InsertAnimalComplaint());
   registerCommand(CommandInsertEmployee, new InsertEmployee());
   registerCommand(CommandInsertFoodComplaint, new InsertFoodComplaint());
   registerCommand(CommandInsertSpecialComplaint, new InsertSpecialComplaint());
   registerCommand(CommandLogin, new Login());
   registerCommand(CommandLoginMenu, new LoginMenu());
   registerCommand(CommandSearchComplaintData, new SearchComplaintData());
   registerCommand(CommandSearchDiseaseData, new SearchDiseaseData());
   ...
  }
  ...
}
```

Figure 3.5: Example of *Forced Join Point*.

Detection Strategy. Based on the characteristics of the *Forced Join Point* anomaly, following detection strategy is proposed:

FJP<pointcut> :=(NAA < LOW **or** LOC_{jp} < LOW) **and**
(LOC_{adv} > HIGH **or** NOP > HIGH)

The metric NAA (*Number of Access Attributes*) counts de number of attributes accessed by the join point. The metrics LOC_{jp} and LOC_{adv} are referring to

the number of lines of code of the join point shadow and the advice, respectively. Finally, the metric NOP (*Number of Parameters*) counts the number of parameters in the pointcut signature.

3.2.2.7. God Aspect

Description. *God Aspect (GA)* occurs when an aspect is realizing[1] more than one concern. In these cases, the aspect could be broken down into as many aspects as the number of concerns it realizes. Therefore, instances of *God Aspect* usually have the following characteristics: (1) they are extensive and complex, (2) they have a high degree of coupling with other modules, and (3) their pointcut expressions and other inner aspect members are not very cohesive. If the measured aspect is heterogeneous and too large, this may indicate that it might be better to decompose the aspect into other aspects. This code anomaly is a specialization of *Large Aspect* (Piveta *et al.*, 2006). *Large Aspect* is just related to a high amount of inner members in an aspect, while *God Aspect* also encompasses the complexity characteristics mentioned above.

Example. Figure 3.6 presents an example of *God Aspect* extracted from the iBATIS system (2009). In this example the aspect named EngineECAspect is responsible for recording contextual information about multiple types of exceptions. However, these exceptions are throwed in the context of different functionalities. To this end, the aspect implementation defines more than 30 members (i.e., attributes, methods, pointcuts, advice, inter-type declarations), which are not cohesive.

[1] refers to when a code element implements partially or completely the concern.

```
aspect EngineCAspect

  public pointcut executionMap(..) : execution(..) && this(..);
  public ErrorContext executionBuildSqlMapEc;

  before(..): executionMap(context) {
    ErrorContext errorCtx = new ErrorContext();
    errorCtx.setActivity("creating the SqlMapClient instance");
    context.setContextObject(..);
    executionBuildSqlMapEc = errorCtx;
  }

  after(..) throwing(Exception e) throws MapException: executionMap(..){
    executionBuildSqlMapEc.setCause(e);
    throw new SqlMapException(..);
  }

  public pointcut executionParseConfig(..) throws ParserException: execution (..) && this(..);
  before(..) : executionParseConfig(..){
    ErrorContext errorCtx = (ErrorContext)context.getContext(..);
    errorCtx.setActivity("creating the ParserClient instance");
    throw new ParserException(..);
  }
 ...
}
```

Figure 3.6: Example of *God Aspect*.

Detection Strategy. Based on the aforementioned characteristics of the God Aspect anomaly, we propose the following detection strategy:

GA:= (CBC > HIGH **and** AS > HIGH **and** LCOO > TWO-THIRDS)

The metric CBC (*Coupling Between Components*) (Sant'Anna *et al.*, 2007) counts the number of code elements with which a given aspect is coupled. The metric AS (*Aspect Size*) counts the number of members defined as part of the aspect implementation. Finally, the metric LCOO (*Lack of Cohesion in Operations*) (Chidamber and Kemerer, 1994; Sant'Anna *et al.*, 2007) counts the number of pairs of methods/advice that do not access at least one attribute in common. Since this metric has a normalized value (between 0 and 1) we used a discrete threshold, two-thirds (0.66). The choice of this metric is derived from its great success in other noteworthy studies related to modularity analysis (Greenwood *et al.*, 2007; Sant'Anna *et al.*, 2007; Figueiredo *et al.*, 2008). Therefore, we considered that it to be a reliable cohesion metric.

It is important to highlight that the presence of a single non-cohesive aspect is not enough to characterize the *God Aspect* anomaly. This can occur also in cases where an aspect encapsulates a heterogeneous crosscutting behavior for the family of join points that it picks out. However, if the aspect is non-cohesive and

too large, it may indicate that could be better to break this module down into multiple aspects.

3.2.2.8. Composition Bloat

Description. *Composition Bloat (CB)* is a complex base computation (e.g. long methods or constructors) that is advised by multiple aspects and, as a result, leads to complex advice implementations in one or more aspects. The symptoms of this code anomaly can be characterized by the complexity of the pointcut (and/or advice) *and* the number of aspects that match the same join point. For instance, whenever complex join points (e.g. methods with long signatures) are shared and affected by multiple aspects, they promote non-trivial interaction between aspects. This phenomenon may be a candidate of accidental complexity, when for example, each of these pointcuts are interested in different information associated with the complex join point (e.g. both client-server information, specific parameters of a long signature, etc). In such cases, the implementation of the base computation or the corresponding advice may be broken down, decreasing the number of aspects sharing the same join point and potentially simplifying their complexity.

Example. The concrete example presented in Figure 3.7 is derived from an optional feature problem extracted from the AspectualMedia system. The MainMidlet.startApp() method is a joint point shared and picked out by several aspects (e.g. PhotoAspect, MusicAspect, VideoAspectand and SMSAspect). The startApp advice is part of the implementation of the PhotoMusicVideoAspect aspect. This aspect tries to modularize the information shared by the Photo, Music and Video concerns. Although this aspect, at first glance, might fall in the anomaly category of *God Aspect*, this is not the case. It contains only a few pointcuts and inner aspect members. Even though the pointcut expression is not complex – one of the possible characteristics of *Composition Bloat* – this aspect is still characterized as an instance of this anomaly. The issue is that the relationship between multiple concerns within the aspect increases the internal complexity of the advice code. The code inspection revealed that the shaded code should be moved to existing aspects already in charge of realizing the corresponding

concerns. For instance, the lines that implement the Photo feature can be defined as part of the implementation of the aspect PhotoAspect.

```
public aspect PhotoMusicVideo{
  ...
  pointcut startApp(MainMidlet mdlt): execution(public void MainMidlet.startpp())
  after(MainMidlet mdlt): startApp(mdlt){
    AlbumData imgModel = mdlt.imageModel;
    AlbumData musicModel = mdlt.musicModel;
    BaseController vCtr= mdlt.videoRootController;
    ... // 13 lines of code removed
    selectcontroller.setMusicController(mCtr);
    selectcontroller.setVideoAlbumData(videoModel);
    mainscreen.append("Photos");
    mainscreen.append("Music");
   mainscreen.append("Videos");
 ...
  }
}
Legend:
☐ Photo      ☐ Music      ☐ Video
```

Figure 3.7: Example of *Composition Bloat*.

Detection Strategy. Based on the characteristics described above the detection strategy is defined as follows:

CB<pointcut> := (NAsJP > HIGH) **and** (LOC_{adv} > VERY_HIGH **or** NOP > HIGH)

The metric NAsJP (*Number of Aspects referring to shared Join Point*) counts the number of aspects that pick out the same set of join points. The metric LOC_{adv} counts the number of lines of code in the advice implementation. Finally, the metric NOP counts the number of parameters of the pointcut signature

3.3. Experimental Evaluation

This section presents the evaluation of the code anomalies in the context of three software systems. Following Wohlin *et al.* suggestion (2000), we defined our study and its goals using the *Goal Question Metric* (GQM) format (Basili *et al.*, 1994), as:

Analyze: aspect-oriented code anomalies

For the purpose of: understanding

With respect to: whether, to what extent, and how (e.g. isolated or simultaneously) they manifest themselves in software systems implementation

From the viewpoint of: the system developers and researchers

In the context of: three (03) software systems from different domains, implemented using different design principles in mind, and by different developers teams.

3.3.1. Target Systems

The first major decision that had to be made in our study was the selection of the target applications. These systems should meet a number of relevant criteria for our study, which are listed in Table 3.1.

Table 3.1: Criteria used for the selection of target systems.

	The target system:
C1	has an implementation with a rich set of aspects, such as implementations of design patterns and widely-scoped crosscutting concerns.
C2	has size (varying from 30 to 60 KLOC)
C3	has an implementation with relevant complexity in terms of number of aspects.
C4	was implemented by different programmer teams with different levels of AspectJ skills.
C5	was developed with modularity and changeability principles as driving design criteria.
C6	has a significant lifetime, comprising several releases.
C7	has its developers available to validate the identified code anomaly instances

Criteria C1 and C3 allow identifying code anomaly in software systems with vary implementation. Criterion C2 enables an in-depth analysis of code anomalies, as required for this kind of study. Criterion C4 allows analyzing whether (or not) code anomalies are specific for developers teams or levels of AspectJ skills. Criterion C5 avoids the introduction of "noise" in the results because designs modeled without taking into consideration good practices are not considered. Criterion C6 supports the observation of the code anomaly behavior throughout the system evolution. Finally, C7 ensures the validation of the identified code anomalies in order to carry out a feasible and reliable analysis.

Based on the aforementioned criteria, 18 releases of 3 medium-size systems were selected. The first one, called iBATIS, is an open source framework for data mapping. It was first released in 2002, and four (04) releases of an aspect-oriented implementation can be found at the SourceForge.net repository (iBATIS, 2009). All the four (04) releases of iBATIS were considered in this study. The second, called Aspectual Health Watcher (Soares *et al.*, 2002; Greenwood *et al.*, 2007; AW, 2009), is a Web-based information system that allows citizens to register

complaints about health issues in public institutions. It was first released in 2000 and 10 other releases are also available at (AW, 2009). All 10 releases of Aspectual Health Watcher were considered in this study. The last system is a product line for deriving applications that manipulate photos, videos and music on mobile devices called Aspectual MobileMedia (Figueiredo *et al.*, 2008). It was first developed in 2004 and 8 releases are available at SourceForge.net repository. Four (04) releases were considered due to widely-scoped changes that were realized in them. A change was considered to be widely scoped when classes and/or aspects underwent many changes across all the systems versions and/or many program elements were added. Table 3.2 lists the analyzed crosscutting concerns for each target system. The characteristics of each target system are presented in Appendix A.

Table 3.2: Analyzed concerns in target systems.

Target System	Crosscutting Concerns
iBATIS	Concurrency, Exception Handling, Type Mapping, Connection, Transaction, Error Context, and 4 Design Patterns
Aspectual Health Watcher (AW)[2]	Concurrency, Exception Handling, Distribution, Persistence, and 5 Design Patterns
Aspectual Mobile Media (AM)[3]	12 Product-Line Concerns: Capture, Controller, Copy, Favorites, Labeling, Media, Music, Persistence, Photo, SMS, Sorting, and Video

3.3.2. Study Phases and Assessment Procedures

After the selection of the target applications, the study encompassed three major phases including: the identification of documented code anomalies in the target systems; the identification of the proposed code anomalies; and the analysis of code anomalies through system evolution.

Identification of Documented Code Anomalies. In order to detect the documented code anomalies in the target systems, we used the conventional metrics proposed by Srivisut and Muenchaisri (2007) as well as those proposed by Piveta *et al.* (2006). In addition, we did not have to adjust the thresholds related to these metrics because they were Boolean values. These metrics were collected

[2] Aspectual Health Watcher is refer to from now on as Aspectual Watcher.

[3] Aspectual Mobile Media is refer to from nowon as Aspectual Mobile.

manually due to the lack of available automated tools supporting them. However, in order to avoid code reviewer fatigue and potential errors in the collection, during sixty (60) days two hours of work per day were dedicated to perform this task. A checklist of review steps was produced in order to guide the process. For instance, all the results were double-checked and all the measures for reviewed using visual inspection to detect false positives. The false negatives were computed using a reference list of actual anomalies recorded and double-checked by the application developers.

Identification of New Code Anomalies. The identification of the proposed code anomalies was also driven by detection strategies and code review. As expected, the measures (and respective detection strategies) led to well-known false positives and false negatives. A code review stage was dedicated to evaluate the accuracy of the detection strategies and overcome their limitations. This stage was based on a systematic visual inspection. We exhaustively analyzed all the cases of false positives and false negatives and needed more time (approx. 90 days) to discover the new code anomalies, as they occurred far more often than the already documented anomalies.

Analysis of Code Anomalies. The goal of the third phase was to trigger some insights for helping maintainers to understand how often the anomalies manifest themselves in software systems. The impact of these anomalies on software architecture as well as which of them are usually removed are further investigated in Chapter 4.

3.3.3. Findings on Code Anomaly Occurrences

There was a significant difference on how often each investigated code anomaly occurred in the target systems. We have analyzed the frequency of both already documented and new code anomalies. The results are summarized in Tables 3.3, 3.4 and 3.5, which show the frequency of anomaly occurrences in the releases of each target system. The new code anomalies are highlighted in the tables with the symbol '*'. The "*Occ*" column indicates the number of times each code anomaly occurred in each release. The "*Occ %*" column presents the percentage of the anomaly occurrence in terms of the total number of all anomaly

occurrences in a particular release. The "*Total*" column presents the total number of anomaly occurrences detected in all the analyzed releases of a system.

Table 3.3: Code anomaly occurrences in iBATIS.

Code Anomaly	iBATIS 1.0		iBATIS 1.3		iBATIS 1.5		iBATIS 2.0		Total
	Occ	Occ %	Occ	Occ %	Occ	Occ %	Occ	Occ %	
Anonymous Pointcut	0	0.0%	1	1.6%	101	64.3%	104	60.0%	216
Redundant Pointcut*	9	27.3%	24	38.1%	25	15.9%	40	21.1%	98
God Pointcut*	9	27.3%	16	38.1%	10	6.4%	13	6.8%	48
Idle Pointcut*	3	9.1%	7	11.1%	9	5.7%	8	4.2%	27
Lazy Aspect	6	18.2%	7	11.1%	2	1.3%	3	1.6%	18
God Aspect*	0	0.0%	1	1.6%	6	3.8%	8	4.2%	15
Composition Bloat*	1	3.0%	1	1.6%	3	1.6%	4	2.2%	9
Duplicated Pointcut	1	3.0%	1	3.2%	2	1.3%	2	1.1%	7
Forced Join Point*	2	6.1%	1	1.6%	1	0.6%	1	0.5%	5
Total	**31**	**100%**	**60**	**100%**	**159**	**100%**	**193**	**100%**	**443**

Table 3.4: Code anomaly occurrences in Aspectual Watcher.

Code Anomaly	AW 1.0		AW 4.0		AW 7.0		AW 10.0		Total
	Occ	Occ %	Occ	Occ %	Occ	Occ %	Occ	Occ %	
Anonymous Pointcut	12	46.2%	27	44.3%	27	40.3%	27	40.3%	93
Redundant Pointcut*	6	23.1%	18	29.5%	18	26.9%	18	26.9%	60
God Aspect*	2	7.7%	4	6.6%	7	10.4%	7	10.4%	20
God Pointcut *	1	3.8%	4	6.6%	6	9.0%	6	9.0%	17
Forced Join Point*	3	11.5%	4	6.6%	4	6.0%	4	6.0%	15
Duplicate Pointcut	2	7.7%	2	3.3%	2	3.0%	2	3.0%	8
Lazy Aspect	0	0.0%	1	1.6%	2	3.0%	2	3.0%	5
Composition Bloat*	0	0.0%	1	1.6%	1	1.5%	1	1.5%	3
Total	**26**	**100%**	**61**	**100%**	**67**	**100%**	**67**	**100%**	**221**

Table 3.5: Code anomaly occurrences in Aspectual Media.

Code Anomaly	AM 4.0		AM 5.0		AM 6.0		AM 7.0		Total
	Occ	Occ %	Occ	Occ %	Occ	Occ %	Occ	Occ %	
Duplicate Pointcut	11	57.9%	13	54.2%	20	54.1%	28	60.9%	72
Lazy Aspect	5	26.3%	3	12.5%	6	16.2%	7	15.2%	21
God Aspect*	1	5.3%	3	12.5%	5	13.5%	4	8.7%	13
God Pointcut *	1	5.3%	2	8.3%	4	10.8%	4	8.7%	11
Composition Bloat*	0	0.0%	1	4.2%	1	2.7%	2	4.3%	4
Redundant Pointcut*	1	5.3%	2	8.3%	0	0.0%	0	0.0%	3
Forced Join Point*	0	0.0%	0	0.0%	1	2.7%	1	2.2%	2
Total	**19**	**100%**	**24**	**100%**	**37**	**100%**	**46**	**100%**	**126**

Notice that a code anomaly can be observed in a release, but addressed by programmers only in the next. Some code anomalies might have a longer life time and be removed only in later releases. In an extreme case, they might have never been removed until the latest release. Therefore, we included in the count for a certain release (Tables 3.3, 3.4 and 3.5), the code anomaly occurrence that

emerged in that release and those reminiscent from previous releases. It was not trivial sometimes to distinguish between both categories in cases where a code element infected by an anomaly suffered significant modifications (e.g. inclusion of new functionalities and refactorings performed). We considered that the anomaly was a new one only if the infected code element had suffered significant structural and semantic modifications. In order to facilitate our analysis, the tables are clustered in two groups, which are separated by dashed lines. The first and second groups contain the code anomalies that occur more and less frequently, respectively. Finally, we opted for not representing the code anomalies that never occurred.

The Variation of Anomaly Occurrences. The total number of code anomalies varied in the target systems. For instance, iBATIS presented the highest number of code anomalies. This happened because the other systems underwent a number of perfective and corrective changes along their longer project history. Therefore, as expected, the consistently-refactored releases of Aspectual Watcher and Aspectual Media implementations would yield less code anomalies than iBATIS. A deeper analysis of the tables shows that the new code anomalies clearly occurred more often than the others in two systems, namely iBATIS (Table 3.3) and Aspectual Watcher (Table 3.4). This result indicates that such anomalies are not specific to a group of developers or particular system characteristics. This is also somehow confirmed in the Aspectual Media case (Table 3.5) as 5 (out of 6) new code anomalies manifested in at least one release.

Code Anomalies that Never Occurred. The anomalies *Abstract Method Introduction*, *Feature Envy*, *Various Concerns* and *Borrowed Pointcut* never occurred in any of the systems. We suspect that this occurs due to the following reasons. First, they might represent silly anomalies that are usually not realized even by programmers with little experience (e.g. *Abstract Method Introduction*). This hypothesis is somehow supported by the fact that they were not encountered even in iBATIS aspects implemented by junior programmers (i.e. who had less than 12 months of experience with AspectJ). Second, it is questionable whether some of these anomalies are really anomalies (e.g. *Various Concerns*). Finally, it might be the case that they have very specific definitions that tend to rarely occur in practice. For instance, *Feature Envy* only considers as foreign members the specific cases of pointcuts defined within a class (allowed in previous versions of

AspectJ). However, other specific members, such as code blocks, should be considered. For example, in some join points classified as *Composition Bloat* it is interesting to move part of the source code from the base code to the aspect. We consider that the definition for Feature Envy (Piveta *et al.*, 2006) should be extended in order to consider such cases.

Code Anomalies with the Highest Frequencies. The code anomalies associated with the problem of replicated pointcuts (e.g. *Redundant Pointcut* and *Duplicate Pointcut*) always fell in the first group considering all target systems. For example, *Redundant Pointcut* accounts for about 20% and 30% of all anomalies in iBATIS and Aspectual Watcher, respectively and *Duplicate Pointcut* accounts for about 56% in Aspectual Media. We also observed that these duplications increased along the releases. A careful analysis made us suspect that this probably occurred because the aspects in those systems were implemented by several developers, who did not know the details about the pointcut expressions defined by others. This finding suggests that the occurrence of *Redundant* or *Duplicate Pointcuts* represents a threat to the system maintenance, as it is likely to trigger ripple effects when pointcut descriptions need to be revisited. *Anonymous Pointcut* (Section 3.2.1) considerably increased along system releases too. We suspect that this occurred due to specific programming styles. In the Aspectual Watcher system, for example, the pointcut expressions were neither complex nor large. They also tended to be referenced by a single advice. We believe that, in this context, these pointcuts should not be regarded as a code anomaly instance, since their expressions are not being reused in other contexts. Differently, the *Anonymous Pointcut* expressions in iBATIS are more complex and some of them were also classified as *Duplicate* and *God Pointcuts* (Section 3.2.2).

Early vs. Late Code Anomalies. We also observed that code anomalies tended to appear in different stages of the software projects. This is an indication that the occurrence of code anomalies largely depends on the software evolution. That is, some code anomalies tend to appear in preliminary releases while others just emerged in later releases. For example, *Composition Bloat* tends to appear in later stages due to the incremental addition of aspects that pick out the same join point. For example, the number of pointcuts that pick out the join point startApp (Figure 3.7) increases along the releases. Furthermore, the same situation occurs with *God Pointcut*, which increases due to the incremental addition of classes and

methods in the base code that should be picked out by a pointcut. On the other hand, we can observe that occurrences of *Lazy Aspect* and *Duplicate Pointcut* tended to appear since the first system releases (Tables Table 3.4 and 3.5).

This finding provides insights into the code anomaly occurrences that can be useful for programmers and software maintainers. For example, the incremental addition and extension of pointcuts that pick out the same set of join points can alert the need to define them in a generic way (Section 3.2.3). Furthermore, the addition of aspects that pick out the same join points, but each of them interested in diverse contextual information (Section 3.2.3), can alert possible occurrences of *Composition Bloat*. This seems to be an indication that the use of detection strategies for aspect-oriented anomalies, which rely on historical information of evolving aspect code (Gîrba *et al.*, 2004, Marinescu *et al.* 2004), can be more effective than detection strategies based on static analysis of a single code version of the system.

Simultaneous Occurrence of Code Anomalies. Table 3.6 presents the identified simultaneous occurrences of code anomalies in the target systems. Each line indicates which pair of code anomalies occurred in one or more systems. In some cases, the same anomaly co-occurrences manifested in more than one system. The last column indicates how many co-occurrences were found in the systems. It is important to highlight that these anomalies did not always occur together. In fact, the number of single instances is higher or equal than the number of simultaneous occurrences. For example, we observed that single instances of *Duplicate* or *Redundant Pointcut* also manifested themselves in a separate fashion through all the systems. The same applies to all the other cases, including instances of *Idle Pointcut* and *Forced Join Point* that occurred in iBATIS and Health Watcher systems.

Table 3.6: Simultaneous occurrences of code anomalies.

System	Code Anomalies							Total
	AP	CB	DP	FJP	GP	IdP	RP	
AW			*	*				4
AW/iBATIS	*						*	31/68
iBATIS	*				*			27
iBATIS		*	*				*	3
iBATIS	*					*		9
iBATIS/AM			*		*			7/3
AM		*	*					3
AM			*	*				2

There was an actual casual connection of *Composition Bloat* to both *Redundant and Duplicate* pointcuts for the vast majority of the 16 instances. It was often the case that a *Composition Bloat* instance was replaced by another instance in further releases. For example, all the pointcuts that pick out the join point startApp (Figure 3.7) were identified as *Redundant Pointcut* instances. These duplications are even more aggravating because they are scattered across several aspects. This leads to ripple effects in the software maintenance as the changes are not localized and it is possible to miss an important change.

Similar situations were observed with respect to *Forced Join Point* and *Idle Pointcut*. For example, in Aspectual Media and Aspectual Watcher, some *Forced Join Point* occurrences were classified as *Redundant Pointcut* and *Duplicate Pointcut*, respectively. In addition, simultaneous occurrences of *Anonymous Pointcut* and *Redundant Pointcut* were observed in Aspectual Watcher and iBATIS systems. Many occurrences of the *Anonymous Pointcut* in iBATIS were also classified as *God Pointcut*. In addition, we can observe that *Redundant* and *Duplicate Pointcut* are usually accompanied by other code anomalies. This observation indicates that duplication of pointcuts often leads to other code anomalies.

3.3.4. Threats to Validity

This section discusses the threats to validity according to the classification proposed by Wohlin *et al.* (2000).

Construct Validity. First, threats to construct validity are mainly related to possible errors introduced in the identification of code smell instances. We are

aware that detection strategies, manual inspection and new metrics can introduce imprecision. However, we limited such threat by using and double-checking results collected from other studies (Soares *et al.*, 2002; Greenwood *et al.*, 2007; Sant'Anna *et al.*, 2007; Figueiredo *et al.*, 2009), such as the automatically-collected measures used in the detection strategies. We have also used code inspection to resolve false positives and false negatives. In addition, at the end, the identified anomalies were all validated with the original developers and the techniques were applied in a way similar to the used in other noteworthy studies (Ferrari *et al.*, 2009; Figueiredo *et al.*, 2008; Figueiredo *et al.*, 2009).

Conclusion Validity. We have two issues that threaten the conclusion validity of our study: the number of evaluated systems and the evaluated AOP mechanisms. We have used in total 18 releases from 3 different systems. Of course, a higher number of systems would always be desired. However, we do believe our sample was appropriate to conduct a thorough investigation of anomaly occurrences, and that it was able to raise hypotheses that can be further tested in replications. Our evaluation was also enough to provide initial statistically-relevant evidence for all hypotheses and findings in the context of an exploratory study. In addition, for each application we ensured that significant changes took place between releases. Related to the second issue, our analysis was concerned with the recurring mechanisms of most AOP languages, such as pointcut–advice and inter-type declarations.

Internal and External Validity. The main threats to internal and external validity are the following. First, the level of experience of the developers in system implementations could be an issue. We used systems that were developed by more than 30 programmers with different levels of AOP skills. The main threat to external validity is related to the nature of the evaluated systems. We have tried to use applications with different sizes and that were implemented using different methodologies and environments. However, we are aware that more studies involving a higher number of systems should be performed in the future.

3.4. Summary

The analysis of code anomalies that infect software systems implementations is a relevant research field since it allows developers to be aware about possible design problems in their implementations. However, the analysis of aspect-oriented code anomalies is particularly hindered by the fact that existing catalogs are very limited (Section 2.3.4.1). Existing catalogs basically mimic object-oriented anomalies, without documenting recurring misuses of strictly aspect-oriented mechanisms. In addition, there is no knowledge about how often code anomalies manifest themselves in aspect-oriented implementations (Section 2.3.4.1).

To fill this gap, this chapter presented and discussed a series of code anomalies related to the misuse of aspect-oriented mechanisms introduced by developers when implementing software systems. These anomalies were derived from our observations and analysis of potentially-anomalous code structures in several systems, reports in the literature, and our experience in the development of aspect-oriented systems. The documented anomalies and the already published ones were classified in three categories according to the aspect-oriented abstraction of mechanisms they are related to: (i) *anomalous pointcut* definitions, (ii) *aspect definition*, and (iii) *undesirable interdependencies*. For each new code anomaly, concrete examples of its manifestation were provided as well as a detection strategy to support its identification.

Furthermore, the chapter presented an empirical study that aims at investigating the frequency rate of aspect-oriented code anomalies in systems' implementation. This study involved the analysis of 790 aspect-oriented code anomalies, distributed in three (03) software systems. These systems represents different domains and were implemented with different design principles in mind and by multiple developers teams.

Our key findings (Section 3.3.3) suggest that:

- Already published anomalies in aspect-oriented code did not occur as often as claimed or expected. Our analysis also indicated that certain anomalies, which were not reported elsewhere, might occur more often than those well-known anomalies.

- Some code anomalies, such as those related to code duplication, tended to manifest themselves in early software releases while others appeared only in later releases, such as those associated with aspect dependencies.
- When aspects are introduced in later releases, many of the aspect-specific code anomalies are a direct consequence of bad object-oriented design of the base code.

All the aforementioned findings and observations provided initial evidence that many of the assessed code anomalies represent recurrent ways in which developers misuse aspect-oriented abstractions and mechanisms. Therefore, we are able to study the impact of code anomalies, including object-oriented and aspect-oriented implementations, considering system architectures (Chapter 4), which is the main goal of this thesis.