
Pós-Graduação em Ciência da Computação

"Idioms to Implement Flexible Binding Times for Features"

By

**Rodrigo Cardoso Amaral de Andrade**

M.Sc. Dissertation

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, MARCH/2012


Federal University of Pernambuco
Informatics Center
Graduate in Computer Science

Rodrigo Cardoso Amaral de Andrade

# "Idioms to Implement Flexible Binding Times for Features"

*A M.Sc. Dissertation presented to the Informatics Center of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.*

Advisor: *Paulo Henrique Monteiro Borba*

RECIFE, MARCH/2012

*Dedico essa dissertação à minha família.*

# Acknowledgements

Eu gostaria de agradecer o suporte do meu orientador Paulo Borba. Não só na orientação impecável da nossa pesquisa, mas também nas horas de dúvidas sobre: "o que danado eu vou fazer depois do mestrado?". Esse suporte foi fundamental para a conclusão do mestrado e para a tomada de decisões bastante importantes sobre o meu futuro.

Agradeço muito também à minha família: minha mãe, meu pai e meu irmão, pelo apoio e incentivo incondicional em todos os momentos. Sempre pude contar com eles.

No mais, gostaria de agradecer a minha namorada, por estar sempre perto de mim, incentivando e me motivando nos trabalhos e aos meus amigos de mestrado que sempre foram desvelados e benevolentes.

Finalmente, agradeço a FACEPE, CNPq e ao projeto INES – Instituto Nacional de Ciência e Tecnologia para Engenharia de Software – por financiarem minha pesquisa.

*Diz o sábio catalão que três exemplos deve o homem imitar, cada qual vindo de um dos reinos da natureza: 1. dos minerais, deve aprender com a água, que obedece à forma do cálice que a contém; 2. entre os vegetais, deve ser como a orquídea, que cresce à sombra das grandes árvores; e 3. do mundo animal, deve espelhar-se na hiena, que segue os leões e não conhece a fome. Assim deve ser o homem: maleável como a água, prudente como as orquídeas e sábio como as hienas.*

—FRANCISCO GOMES, O CHALAÇA

# Resumo

Empresas têm adotado Linhas de Produtos de Software (LPS) como paradigma de desenvolvimento para obter melhoras significativas no tempo de produção, custos de manutenção, produtividade e qualidade dos produtos. LPS engloba uma família de sistemas intensivos de software que são desenvolvidos a partir de artefatos reusáveis. Com o reuso de tais artefatos, é possível construir um grande número de produtos diferentes aplicando diversas composições. Há uma variedade de técnicas usadas amplamente para desenvolver LPS. Por exemplo, programação orientada a aspectos (POA), programação orientada a *feature* e compilação condicional. Essas técnicas diferem no tipo de composição para criar um produto de uma LPS tanto estática quanto dinamicamente.

Neste contexto, é importante definir quando determinadas *features* devem ser ativadas no produto devido a requisitos específicos de clientes e aplicações em cenários diferentes. Desse modo, o *binding time* de uma *feature* é o tempo em que se decide ativar ou desativar uma *feature* de um produto. No geral, *binding time* estático ou dinâmico são levados em consideração. Por exemplo, produtos feitos para dispositivos que possuem recursos restritos podem usar *binding time* estático ao invés de dinâmico, devido ao *overhead* introduzido por este último. Para dispositivos sem restrições de recursos, o *binding time* pode ser flexível, *features* podem ser ativadas estática ou dinamicamente.

Para prover *binding time* flexível para *features*, pesquisadores propuseram um idioma baseado em AspectJ e padrões de projeto chamado Edicts. A ideia consiste em suportar flexibilidade de *binding time* para *features* de maneira modular e conveniente. No entanto, nós observamos problemas de modularidade no idioma Edicts. Apesar de geralmente nós usarmos aspectos para resolver problemas *crosscutting* comum em classes, esses problemas aparecem nos próprios aspectos. De fato, muitos estudos indicam que esses problemas são ruins para modularidade de software. Desta forma, nós observamos que Edicts clona, espalha e entrelaça código através da sua implementação, o que pode acarretar em tarefas demoradas, como manutenção de código duplicado.

Desta forma, nós desenvolvemos três idiomas e os implementamos para prover *binding time* flexível para *features* de quatro aplicações diferentes. Além disso, nós avaliamos Edicts e os três idiomas quantitativamente através de métricas considerando duplicação, espalhamento, entrelaçamento e tamanho de código, além de tentar garantir que não há alteração de comportamento entre suas implementações.

**Palavras-chave:** Binding Time Flexível; Linha de Produtos de Software; Idiomas; AspectJ; CaesarJ.

# Abstract

Companies are adopting the Software Product Line (SPL) development paradigm to obtain significant improvements in time to market, maintenance cost, productivity, and quality of products. SPL encompasses a family of software-intensive systems developed from reusable assets. By reusing such assets, it is possible to construct a large number of different products applying various compositions. There is a variety of widely used techniques to develop SPLs, such as aspect-oriented programming (AOP), feature-oriented programming (FOP), and conditional compilation. These techniques differ in the type of composition to create a product within the SPL static or dynamically.

In this context, it is important to define when certain features should be activated in the product due to specific client requirements and different application scenarios. Thereby, the binding time of a feature is the time that one decides to activate or deactivate the feature from a product. In general, static and dynamic binding times are considered. For example, products for devices with constrained resources may use static binding time instead of dynamic due to the performance overhead introduced by the latter. For devices without constrained resources, the binding time can be flexible, features can be activated or deactivated statically or users may do it on demand (dynamically).

To provide flexible binding time for features, researchers proposed an AOP idiom based on AspectJ and design patterns named Edicts. The idea consists of supporting binding time flexibility of features in a modular and convenient way. However, we observe modularity problems in the Edicts idiom. Although we usually use aspects to tackle crosscutting concerns common in classes, such a problem now appears within the own aspects. Indeed, several studies indicate that these concerns hurt software modularity. This way, we observe that Edicts clones, scatters, and tangles code throughout its implementation, which may lead to time consuming tasks, such as maintaining duplicated code.

This way, we develop three idioms and implement them to provide flexible binding time for features of four different applications. In addition, we evaluate Edicts and the three idioms quantitatively by means of metrics with respect to code tangling, scattering, cloning, size, and also try to guarantee that our idioms do not change feature code behavior among the different implementations.

**Keywords:** Flexible Binding Time; Software Product Line; Idioms; AspectJ; CaesarJ.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Nowadays, it is primordial to develop software taking into account the client's needs. Hence, companies are offering software increasingly customized. However, developing customized software is expensive but clients/customers do not want to pay such high costs for it. In this context, software reuse [Kru92, JGJ97] has been proposed to avoid repetition of tasks, such as writing code that already has been written. Thus, it allows reduction of cost of development. Nevertheless, it is necessary to plan the reuse of software artifacts in order to make this practice viable.

In this context, companies are adopting the Software Product Line (SPL) development paradigm to obtain significant improvements in time to market, maintenance cost, productivity, and quality of products. SPLs provide means to compose software products that match the requirements of different application scenarios from a single assets base [RSSA08]. It encompasses a family of software-intensive systems developed from reusable assets. By reusing such assets, it is possible to construct a large number of different products applying various compositions [PBvdL05].

Researchers proposed some techniques to develop SPLs, such as *aspect-oriented programming (AOP)* [KLM+97], *feature-oriented programming (FOP)* [Pre97], and *conditional compilation*. These techniques differ in the type of composition to define a product within the SPL. However, these compositions may be defined while the clients are using the product. For instance, the client may need to deactivate some database transactions at peak times to avoid interrupting the complete process. Therefore, developers should provide the binding of these compositions not only when building it, but also while clients use their product. The variation among the products generated by different compositions is represented through features in the SPL context. Features are the semantic units by which different applications within an SPL can be differentiated and defined [TBD06].

In this context, it is important to define when certain features should be activated in the product due to specific client requirements and different application scenarios. Thereby, the binding time of a feature is the time that one decides to activate or deactivate the feature from a product [CRE08]. In general, we consider two different binding times: static and dynamic. For example, products for devices with constrained resources may use static binding time instead of dynamic due to the performance overhead introduced by the latter [RSSA08]. For devices without constrained resources, the binding time can be flexible, features can be activated or deactivated statically or users may do it on demand (dynamically). Besides these reasons, previous works identified the importance of flexible binding time too [vO02, CRE08, RSSA08].

In this work, we are concerned with the flexibility of the aforementioned composition. We use the AOP technique to develop solutions that implement the flexibility of binding time for features to achieve customization of software with the advantages of using the SPL approach.

## 1.1 Problem

To provide flexible binding time for features, researchers proposed an AOP idiom based on AspectJ [KHH+01a] and design patterns [GHJV95] named Edicts. Chakravarthy et al. [CRE08] claims it consists of supporting binding time flexibility of features in a modular and convenient way. However, we observed modularity problems in the Edicts idiom. Although we usually use aspects to tackle crosscutting concerns [KLM+97] common in classes, such a problem now appears within the own aspects. Indeed, several studies indicate that these concerns hurt software modularity [KSG+06, LB05, EZS+08]. This way, we observe that Edicts clones code throughout its implementation, which may lead to time consuming tasks, such as maintaining duplicated code. Code **cloning**, which is a duplication of code fragments, means that bugs can appear in several places, it may lead to poor code readability, and difficulties to make changes. Additionally, Edicts presents code **scattering** and **tangling**, which are evidences that the implementation of a concern is not well modularized [LLO03]. This could hamper the reuse and understanding of the source code. Little reuse and understanding may impact on the productivity of development, for example.

In this scenario, there is no AOP-based idiom to implement flexible binding time that reduce these problems. Hence, we introduce idioms to address these shortcomings presented by Edicts [ARG+11]. Furthermore, there is no previous work in the literature that

evaluate idioms considering modularity by means of metrics. Thus, besides introducing idioms, we need to assure they present better results than Edicts regarding code cloning, scattering, and tangling.

## 1.2 Proposed Solution

Given the problems introduced in Section 1.1, we propose three idioms to implement flexible binding time for features. The idioms are AOP-based, so we use some of its constructs to modularize feature code and implement static and dynamic binding time. We design them with the goal of reducing code cloning, scattering, and tangling presented by Edicts. We implement Edicts and our three idioms to provide flexible binding time into several features of four applications (Tetris [Tet09], Freemind [Fre09], ArgoUML [Arg09] and BerkeleyDB [Ber10]).

In addition, we evaluate Edicts and the proposed idioms with respect to modularity, source code size, and behavioral changes between the feature implementation with the different idioms. To evaluate modularity, we use a metric suite, which encompasses metrics that measure code cloning, scattering, and tangling considering different levels of granularity. For example, we measure code scattering for package, class or aspect, and method or advice levels. We define our idioms to achieve low rates of these metrics, and consequently, better modularity. To evaluate source code size, we use metrics to measure the number of components and source lines of code for a given application. In summary, the evaluation results show that Edicts presents code cloning, scattering, and tangling in contrast to our idioms, which mitigate these problems. Furthermore, we investigate behavioral changes in our implementations. We compare the behavior of the same application implemented with different idioms to detect an eventual change. For example, we compare the execution between a given feature, which we provide flexible binding time with Edicts, and the same feature implemented with other idiom. Ideally, there should not be differences in the feature behavior among these implementations.

## 1.3 Contribution

The main contributions of this work are the following:

1. Propose two idioms based on AspectJ and one idiom based on CaesarJ [AGMO06] to implement flexible binding time for features;

2. Implement Edicts and the proposed idioms in 18 features of four real applications;
3. Evaluate the four idioms quantitatively by means of metrics with respect to code tangling, scattering, cloning, and size;
4. Verify that no behavioral changes is found in the four applications.

## 1.4 Outline

The remainder of this dissertation is organized as follows:

- Chapter 2 reviews essential concepts used throughout this work. Namely, Software Product Lines and Aspect-Oriented Programming;
- Chapter 3 shows the problems found with Edicts and introduces the three proposed idioms to implement flexible binding time. We explain the design and an example of each idiom;
- Chapter 4 presents the study settings which explains each case study, the Goal-Question-Metric approach we used, and the assessment procedure we performed in this work. Additionally, it presents the metric results and the behavior validation for the applications implemented with the four different idioms. Finally, it discuss the results;
- Chapter 5 concludes our work and discusses some related and future work;
- Appendix A presents the implementation of the four idioms for a given feature.
- Appendix B presents the complete metric results.

# 2 Background

In this chapter, we provide details about concepts used throughout this work. First, we introduce Software Product Lines (SPL) in Section 2.1. We use AspectJ and CaesarJ to implement SPLs, so we explain both in Section 2.2.

## 2.1 Software Product Lines

Product Lines allow the development of products using platforms and mass customization [PBvdL05]. Developing platforms means to aim at reuse by building reusable parts. Mass customization focuses on large scale production. Thus, the variability between the products is important to attend the client's requirements.

More specifically, Software Product Lines provide means to compose software applications that match the requirements of different scenarios from a single code base [RSSA08]. The single code base represents the commonalities between the applications within the SPL. In addition, to match these requirements, SPL encompass variabilities, which define different properties for each SPL instance. This way, SPL simplify software reuse because it uses a wide variety of common artifacts to define different applications. For example, we may reuse software requirements, architecture, and tests.

By using SPLs to provide customized applications at reasonable costs, we can achieve improvements on software development [PBvdL05], as follows:

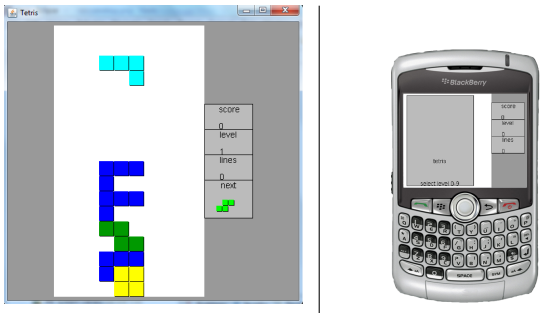
1. **Reduction of development costs.** SPL are meant to reuse artifacts in several different kinds of applications, this implies in cost reduction since we do not develop each application from scratch;

2. **Enhancement of quality.** The common artifacts are reviewed and tested in each product. This enhances the chance to find errors and fix them. Therefore, it

increases the quality of applications;

3. **Reduction of time to market.** Although the time to market is initially higher for SPL, it is shortened after the common parts are built, as many artifacts can be reused for new applications.

To exemplify a Software Product Line, consider our Tetris example. It is possible to generate two different applications from the same base code. Figure 2.1 illustrates two instances of Tetris. The first instance (Figure 2.1(a)) is a Tetris version running for the desktop environment, whereas the second instance (Figure 2.1(b)) runs for the mobile environment. For this example, we first implement the code corresponding to the platform, which is the common part independently of what environment it runs. Next, we implement the variabilities of each application within the SPL, which is the specific code of each environment. Thereby, if we need to create an application to run for an additional environment, we reuse the platform code and implement the variabilities instead of building the application from scratch.




(a) Tetris for desktop.

(b) Tetris for mobile.

**Figure 2.1** Tetris SPL.

In this context, we can map these variabilities in features and use models to describe the commonalities and variabilities of an SPL, beyond the implementation artifacts. Hence, Section 2.1.1 introduces the concept of feature and discusses feature models.

### 2.1.1 Feature Models

Features are increments in application functionality. They are the semantic units by which different applications within an SPL can be differentiated and defined. Different compositions of features yields different instances within an SPL [TBD06].

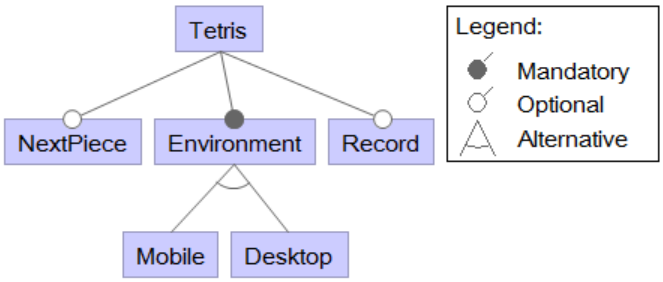



**Figure 2.2** Tetris SPL Feature Model

Different composition of features forms distinct applications within an SPL. Feature Models [KCH+90] outline these compositions in a simple way, depicting the possible commonalities and variabilities among these applications. A Feature Model consists of diagrams, which describe a hierarchical decomposition of features. Features may assume four types in a Feature Model, as follows:

1. **Mandatory.** It is part of the commonalities, so it is present in all applications within an SPL (filled circle);
2. **Optional.** It is part of the variabilities, so it may be present or not (empty circle);
3. **Alternative.** It is part of the variabilities, however it is mutually exclusive within a group of features. Thus, at least one, and only one feature can be selected for a single application (empty arc);
4. **OR.** It is part of the variabilities, as it is freely selected within a group of features, albeit at least one has to be selected (filled arc).

For instance, Figure 2.2 illustrates the Feature Model of our Tetris example. Notice that the **Environment** feature is mandatory, it is present in all possible feature compositions. In addition, **Mobile** and **Desktop** are alternative features, so it means they are mutually exclusive, only one of them can be present in an SPL instance. On the other hand, **NextPiece** and **Record** are optional features. They may be present or not in a feature composition.

## 2.2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [KLM+97] is a programming paradigm proposed to overcome Object Orientation (OO) deficiencies. When using OO, some concerns

are crosscutting. It means that they are tangled and scattered, and consequently hard to understand and maintain. By using AOP, it is possible to separate these crosscutting concerns. Therefore, it modularizes code related to specific functionalities that are scattered and tangled among other parts of the application. Thereby, AOP enhances software modularity. For example, transaction management, persistence, binding time are well-accepted example of such concerns [CRE08, SLB02].

In this context, there are several languages that support AOP. In this work, we focus on two languages (AspectJ and CaesarJ). Thus, we present the first one in Section 2.2.1 and the second one in Section 2.2.2.

### 2.2.1 AspectJ

AspectJ [KHH+01b] extends Java with support to AOP. It modularizes crosscutting concerns using units called aspects, which are associated with classes by a weaving process. Thus, well defined OO concepts are implemented in classes whereas crosscutting concerns are tackled by AspectJ aspects. By defining aspects it is possible to alter or add class members, such as methods. Furthermore, it can alter hierarchy of classes, manipulate exception, and add behavior at specific points. In what follows, we detail the structures that we can define within an aspect.

**Intertype declaration.** It allows the modification of an application's static structure. We may declare class members in a separate aspect, such as methods, attributes, interfaces, and constructors. Then, the aspect associates these members to the existing classes. For example, Listing 2.1 illustrates an alteration of the `TetrisCanvas` class hierarchy (Line 1), and an introduction of the field `nextpiecebox` into the `TetrisCanvas` class (Line 3).

**Listing 2.1** Example of intertype declaration.

```
1 declare parents : TetrisCanvas extends JPanel;
2
3 NextPieceBox TetrisCanvas.nextPieceBox;
```

**Listing 2.2** Pointcut example.

```
1 pointcut pcnextpiecebox() : execution(* TetrisCanvas.createNextPieceBoxHook());
```

**Pointcut.** It is a set of join points, which are points in application execution flow. This way, pointcuts are expressions that match certain join points at run-time. For instance, join points could be method execution, variable access, or object instantiation. Listing 2.2 illustrates a pointcut that defines an expression to match the `createNextPieceBoxHook`

method execution from the `TetrisCanvas` class.

In this work, we use a special pointcut named `adviceexecution`. It picks out a set of advice execution join point. For example, we could intercept the execution of all pieces of advice defined in a specific aspect.

**Advice.** It implements behavior to be added when a join point is reached. The behavior is executed before, after or around a join point. Listing 2.3 implements an advice that instantiates a new `NextPieceBox` after the join point, which is matched by the `pcnextpiecebox` (Listing 2.2), is reached in the execution flow.

**Listing 2.3** Advice example.

```
1  after() : createNextPieceBoxHook() {
2   nextPieceBox = new NextPieceBox();
3  }
```

In this work, we define two AspectJ-based idioms to implement flexible binding time for features. The third idiom is based on CaesarJ, which is another language that supports AOP.

### 2.2.2 CaesarJ

CaesarJ [AGMO06] is an extension of Java [AG96], which supports AOP. It combines pointcut and advice with advanced OO modularization mechanisms, such as virtual class and dynamic aspect deployment. CaesarJ addresses the intertype declaration problem of integrating independent components into an application. Therefore, CaesarJ does not modify the component to be integrated. In what follows, we explain CaesarJ constructors we use in this work.

**Listing 2.4** CaesarJ class example.

```
1  cclass Checksum {
2   ChecksumValidator validator;
3
4   pointcut readHeader() : call(void EntryHeader.readHeader());
5
6   after() throws DatabaseException : readHeader() {
7    validator = new ChecksumValidator();
8    validator.update();
9   }
10 }
```

**CaesarJ class.** It is defined using the keyword `cclass`. Differently from pure Java classes, it also can define aspect constructs, such as pointcut and advice. We can instantiate and reference it as an object. Furthermore, inheritance hierarchies of CaesarJ

and Java classes are strictly separated in CaesarJ. A `cclass` may not inherit from a Java class and vice versa. Listing 2.4 illustrates an example of a CaesarJ class. It defines an attribute (Line 2), a pointcut to intercept calls to the `readHeader` method (Line 4), and an advice to add behavior after calling this method (Lines 6-9).

**Wrapper class.** It is a dynamic extension of other classes, called wrappees. A wrapper class can introduce new state and operations, as well as adapt the wrappee to required interfaces. The wrapper-wrappee relationship is established by the keyword `wraps`. A wrapper can access its wrappee by means of the special identifier `wrappee` [AGMO06]. Hence, we are able to separate different concerns by means of wrapper classes. In AspectJ, we could do this by using intertype declarations. Listing 2.5 defines the `FileReaderW` wrapper class. It implements the `startChecksum` method. This method calls the `threadSafeBufferPosition` method, which is defined in the `FileReader` class from the base code, by the keyword `wrappee`.

**Listing 2.5** Wrapper class example.

```
1 cclass FileReaderW wraps FileReader {
2  void startChecksum() {
3   wrappee.threadSafeBufferPosition();
4 }
```

**Aspect deployment.** We can instantiate CaesarJ classes equally to conventional object aspects by using the keyword `new`. However, instantiation does not activate its pointcuts and advice because CaesarJ classes are not deployed by default. We may explicitly deploy them statically or dynamically. Listing 2.6 and 2.7 illustrate the `Checksum` class (Listing 2.4) static and dynamic deployment, respectively. Notice that using the keyword `deployed`, the `Checksum` class is deployed statically. On the other hand, when using the keyword `deploy`, the `Checksum` class is deployed dynamically. This allows us to control whether the feature code is executed either statically or dynamically.

**Listing 2.6** Static deployment example.

```
1 deployed cclass ChecksumStatic extends Checksum {
2 }
```

**Listing 2.7** Dynamic deployment example.

```
1 deployed cclass ChecksumValidatorAspect {
2  pointcut pc_jarmain() : execution(* JarMain.main(..));
3  before() : pc_jarmain() {
4   if (isActivated("checksum"))
5    deploy new Checksum();
6  }
7 }
```

# 3

# Flexible Binding Time

In this chapter, we depict problems we found in an existent solution for supporting flexible binding time and we introduce our idioms to reduce these problems. We discuss the motivation for supporting flexible binding time in Section 3.1. Then, we present our three idioms, detailing its design and illustrating how it works in a running example in Section 3.2. We introduce these idioms to address the shortcomings we found in Edicts. Each idiom improves the one introduced previously in some way, such as less code scattering or tangling. Later in Chapter 4, we evaluate these idioms regarding to code cloning, scattering, tangling, size, and behavior. In the end, we discuss how to apply our idioms in the presence of feature interaction. This happens when a given feature code we want to provide flexible binding time is part of another feature code (Section 3.3).

## 3.1 Motivation

To implement flexible binding time, we could use the Edicts idiom [CRE08], which makes it possible to choose between compile (static) and run-time (dynamic) binding for features. The basic idea is to modularize feature implementation by means of AspectJ [KHH+01a] aspects. The programmer applies the Edicts idiom using separate aspects for static and dynamic binding, as illustrated in Figure 3.1. The `AbstractAspect` contains intertype declarations to modularize the part of a feature implementation that we do not need to deactivate because they are called only from within the own feature implementation, therefore this code is not executed if the feature is deactivated. The implementation of intertypes in the abstract aspect avoids duplicating feature code among the concrete subaspects. However, we may need to deactivate some intertype declarations in specific situations by introducing driver code. For example, a mandatory feature could call a method that is implemented differently among a group of alternative features. There-

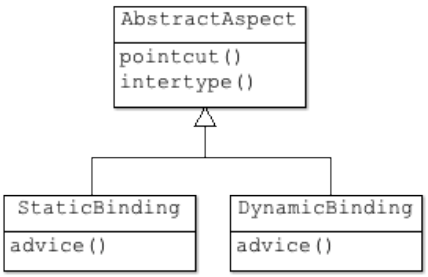



**Figure 3.1** The structure of Edicts [CRE08]

fore, the call to this method always exist in the base code, albeit its implementation is modularized within the feature implementation. So, we need to introduce driver code in the intertypes for such situations. Additionally, the `AbstractAspect` contains concrete pointcuts that target the variation points, which are points in the base code where the feature code should be introduced when activated. The `StaticBinding` and the `DynamicBinding` are concrete subaspects that implement static and dynamic binding time, respectively. They differ because the `DynamicBinding` aspect implements a driver mechanism, for example an `if` statement, to dynamically decide whether the feature code should be executed, but both contain advice declarations that alter system behavior by adding or changing feature functionality into the join points identified by the pointcuts defined in the `AbstractAspect`. Note that the intertype declarations aforementioned are referenced within these pieces of advice. Hence, they are executed only when these pieces of advice are activated.

Despite of separating feature code and supporting flexible binding time, Edicts may lead to a number of modularity issues, including code **cloning**, **tangling**, and **scattering**. To illustrate these problems, consider the *Checksum* optional feature from BerkeleyDB, which is an open-source database written in Java. This feature implementation detects data corruption when writing or reading a database page. Following the structure of Edicts (Figure 3.1), we have concrete pointcuts and intertypes in abstract aspects, and advice declarations in concrete subaspects. However, this design may lead to code **cloning** because we duplicate the pieces of advice among the concrete subaspects to implement static and dynamic binding time. For the *Checksum* feature, we would end up cloning 17 pieces of advice. Listing 3.1 and 3.2 show part of the cloned code. The only difference appears in Lines 4 and 9 from Listing 3.2. These lines implement the aforementioned driver mechanism. Hence, the code between Lines 5 and 8 is executed only if the driver activates the feature, which can be based on an user decision, for example. For

simplicity, we do not show the `ChecksumAbstract` implementation, which declares concrete pointcuts and intertypes related to the *Checksum* feature and referenced by the concrete subaspects. Section A.1 from Appendix A contains the Checksum complete implementation using Edicts.

In our experience, the presence of advice is very common in feature implementation modularized by aspects. It is common because most of the feature implementations are tangled with other concerns in method level. Thus, we often need to extract part of a method code to a piece of advice [KAB07, MLWR01]. For instance, only one of our 18 implementations of features from four case studies does not have advice. In fact, applying Edicts contributes to clone each piece of advice among the concrete subaspects of 17 features, resulting in 177 pieces of advice cloned in our studies, we provide further details in Section 4.2.1.

**Listing 3.1** Checksum static binding.

```
1  aspect ChecksumStatic extends ChecksumAbstract {
2    ...
3    after() throws DatabaseException : readHeader() {
4      if (lm.doChecksumOnRead) {
5        validator = new ChecksumValidator();
6        ...
7      }
8    }
9    ...
10 }
```

Besides the cloning problem, the code in Listing 3.2 does not have only feature implementation, it is **tangled** with the code of the aforementioned driver mechanism. This may lead to increased complexity of addition, removal or modification of driver code [CHOT99]. For example, we may need to change the driver mechanism due to a new configuration of the environment where the application is running, such as less memory resources. In this scenario, we would have to add the new driver condition into each piece of advice. Moreover, the situation can get even worse when applying Edicts to large features that require many advice declarations, since we tangle this mechanism code with feature implementation for each piece of advice. Even if we use an interface to reference the driver within the pieces of advice, we could still have a problem because these interfaces may vary as well. This way, when using Edicts, we apply aspects to modularize feature code, but we do not use aspects to modularize the driver mechanism code.

Furthermore, `if` statements like the one in line 4 (Listing 3.2) are **scattered** throughout the pieces of advice to support dynamic binding time. Such code scattering is

error-prone, because forgetting an `if` statement may raise run-time exceptions in the application, such as `NullPointerException`. Changing the driver mechanism is also time consuming because we have to alter driver code within each piece of advice.

The cloning issue could be mitigated by writing the feature code as methods in a separate aspect or class, which is not part of the application's logic. Indeed, we could decrease code cloning by duplicating only advice declarations, and calling these methods in the pieces of advice. Thus, the cloning problem would be reduced but not eliminated. Besides that, this solution might often not work since AspectJ does not support `proceed` calls outside advice. Thus, we could not call it from within the methods in a separate aspect. Further, we could not pass parameters through these absent `proceed` calls, which could lead to inconsistent object states. For instance, if we alter an object in these methods that is used in the base code, we could not access its new state in the base code. If we create a separate class instead of an aspect to implement the aforementioned methods, we would worsen the problem because classes do not support *privileged* access to non-public members, so we would have to change the visibility of non-public methods called within the advice body.

**Listing 3.2** Checksum dynamic binding.

```
1  aspect ChecksumDynamic extends ChecksumAbstract {
2   ...
3   after() throws DatabaseException : readHeader() {
4    if (driver.isActivated("checksum")) {
5     if (lm.doChecksumOnRead) {
6      validator = new ChecksumValidator();
7      ...
8     }
9    }
10  }
11  ...
12 }
```

Furthermore, this design to mitigate the cloning issue would increase code scattering because it uses an additional component that contains feature code. For example, the **Checksum** feature implementation would have four aspects instead of three. The feature code would be even more scattered when its implementation uses more aspects. Nevertheless, the tangling would be the same because the `if` statements (Line 5, Listing 3.2) would still be implemented in the same way, as we would only extract feature code.

As the problems just presented might harm software modularity, we propose three idioms to address the Edicts shortcomings. The first idiom address the Edicts shortcomings we just presented. The second idiom address the problems of Edicts and also some issues of our first idiom. The third idiom reduces the problems even more and is

not AspectJ-based. Moreover, we evaluate these idioms in Chapter 4 and discuss these problems more specifically throughout this work.

## 3.2 Idioms

In this section, we describe our idioms to provide flexible binding time for features with the aim of mitigating the discussed problems with Edicts. As a consequence, we have less code cloning, tangling, and scattering. In Section 3.2.1, we introduce the Pointcut Redefinition idiom. It avoids cloning pieces of advice as well as reduces feature and driver code tangling and scattering. However, this idiom may increase the implementation size because it redefines the pointcuts related to the feature. Hence, we describe the Layered Aspects idiom in Section 3.2.2. It also addresses the discussed problems with Edicts, but the implementation size and the driver scattering are smaller. Additionally, we introduce the Flexible Deployment idiom in Section 3.2.3 to evaluate idioms implemented with different techniques. Unlike the others, this idiom is based on CaesarJ, which allows an improvement regarding the discussed problems, but on the other hand, it does not work in some cases that we describe in Section 4.2.7.

### 3.2.1 Pointcut Redefinition

The Pointcut Redefinition idiom uses inheritance of AspectJ aspects and redefinition of pointcuts to provide binding time flexibility. Essentially, we modularize the feature code, either advice or intertypes, in an abstract aspect. To implement static binding, we define an empty concrete subaspect to permit feature code instantiation by inheriting the abstract aspect. By compiling the application with this concrete subaspect, we activate all intertypes and advice declarations, so the application execution will run the feature code. On the other hand, if we do not compile this concrete subaspect, the application execution will not run the feature code. For dynamic binding, we define another concrete subaspect that redefines the pointcuts from the abstract aspect restricting them with the driver mechanism, that is, the new pointcuts we define in this aspect. This way, when compiling the application with this concrete subaspect, we are able to dynamically decide whether the feature code is executed. In what follows, we provide details about this idiom.

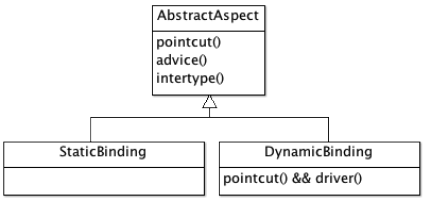



**Figure 3.2** The structure of Pointcut Redefinition

### Design

Pointcut Redefinition is implemented using AspectJ aspects. We define an abstract aspect which may contain advice, pointcuts, and intertypes related to the feature code modularization. Figure 3.2 illustrates an overview of Pointcut Redefinition's structure.

This idiom provides static binding by an empty concrete subaspect to allow the `AbstractAspect` instantiation and the feature code execution. For dynamic binding, we redefine the `AbstractAspect` concrete pointcuts and associate them with the driver code, which is a mechanism (or a set of mechanisms) responsible for providing information about whether a feature should be executed at run-time, we show an example in Listing 3.4 of Section 3.2.1. This way, the pointcuts defined in the `AbstractAspect` intercept the corresponding join points only if the driver activates the feature.

Notice that we apply the driver to advice and pointcuts. However, we may need to apply the driver for intertypes in specific situations, as we do with Edicts. For example, a mandatory feature could call a method that is implemented differently among a group of alternative features. This way, the call to this method always exist in the base code, albeit its implementation is modularized within the feature implementation. Therefore, in such situations, we introduce driver code in the intertype declarations as an `if` statement similarly to Edicts. Nevertheless, our case studies do not present this scenario in any feature.

In addition, notice that static and dynamic binding time do not coexist at the same product. We include only the `StaticBinding` aspect or the `DynamicBinding` aspect in the compilation with the `AbstractAspect`. Therefore, we provide three different variability possibilities: feature dynamically bound or unbound, feature statically bound and, feature statically unbound. This also applies to our idiom presented in Section 3.2.2.

The discussed structure of Pointcut Redefinition avoids advice code cloning because there is no need to duplicate pieces of advice with feature code among the concrete subaspects because they are in the abstract aspect. Moreover, it solves the tangling

between driver and feature code because the driver mechanism is implemented in a separate subaspect (`DynamicBinding`). There is no feature code in the concrete subaspects and no driver code in the abstract aspect, so they are not scattered among the three aspects. However, the redefinition of pointcuts may increase the idiom's implementation size when several pointcuts are present. This also leads to scattering of driver code.

### Example

To explain the Pointcut Redefinition idiom more clearly, we use the same example from Section 3.1, the *Checksum* optional feature. In addition, Section A.2 from Appendix A contains the Checksum complete implementation using this idiom.

**Listing 3.3** ChecksumAbstract.

```
1  abstract aspect ChecksumAbstract {
2   ...
3   pointcut readHeader() : call(void EntryHeader.readHeader());
4   ...
5   pointcut addPrevOffset(int entrySize)
6    : execution(ByteBuffer LogManager.addPrevOffset()) && args(entrySize);
7   ...
8   after() throws DatabaseException : readHeader() {
9    if (doChecksumOnRead) {
10    validator = new ChecksumValidator();
11    ...
12   }
13  }
14  ...
15  ByteBuffer around(int entrySize) : addPrevOffset(entrySize) {
16   return proceed(entrySize);
17  }
18  ...
19  void FileReader.validateChecksum() {
20   ...
21  }
22 }
```

**Feature implementation.** The abstract aspect contains intertype and advice declarations as well as the pointcuts related to the *Checksum* feature code. Listing 3.3 shows a simplified abstract aspect that implements the aforementioned elements. Lines 3 and 5 define pointcuts that match join points where the feature code should be introduced. The pieces of advice defined between Lines 8 and 17, contain feature code that we introduce in the join points matched by these pointcuts. Moreover, Lines 19-21 define an intertype declaration that introduces the `validateChecksum` method in the `FileReader` class. In contrast to the code in the pieces of advice, the code of this method should not

be introduced at a given join point, the own method is referenced by another member in the feature implementation, therefore we use intertype declaration.

To avoid cloning the pieces of advice, we implement them in the abstract aspect differently from Edicts. We may have as many aspects as needed to modularize code related to the feature, this is an engineering decision. Nevertheless, we apply the structure illustrated in Figure 3.2 for each aspect created.

**Implementing static binding time.** As mentioned in Section 3.2.1, we create an empty concrete subaspect to allow the `ChecksumAbstract` instantiation. To activate the feature statically, we include the `ChecksumAbstract` and `ChecksumStatic` aspects in the project build. To deactivate the feature statically, we do not include any of these aspects.

**Implementing dynamic binding time.** Now we are ready to define the aspect responsible for dynamic binding time. In Line 3 of Listing 3.4, we show the driver implementation. In this case, we check if the checksum property corresponds to `true` or `false` in a properties file. If this property's value corresponds to `true`, the feature is activated and its code should be executed. However, we may need to implement the driver in a different way. For example, we could have many different drivers, which could lead to complex boolean expressions. Furthermore, the driver mechanism does not need to obtain information about whether the feature code should be executed on properties file. It could vary from GUIs that ask the user if he/she wants to activate the feature to more complex ones like sensors that decide by themselves. In Lines 5-8, we redefine the pointcuts defined in `ChecksumAbstract` and associate the driver mechanism. This way, the driver controls whether the pointcuts are applied dynamically. If the feature is activated, the pointcuts are applied and consequently the code within the pieces of advice is executed. On the other hand, if the feature is deactivated, the pointcuts are not applied and the feature code is not executed.

**Listing 3.4** ChecksumDynamic.

```
1 aspect ChecksumDynamic extends ChecksumAbstract {
2
3   pointcut driver(): if(new Driver().isActivated("checksum"));
4
5   pointcut readHeader() : ChecksumAbstract.readHeader() && driver();
6
7   pointcut addPrevOffset(int entrySize)
8     : ChecksumAbstract.addPrevOffset(entrySize) && driver();
9 }
```

To observe the Pointcut Redefinition disadvantages, notice that when we define several pointcuts, the dynamic binding implementation may increase its size because we redefine

all pointcuts. Additionally, we scatter the driver throughout the redefined pointcuts, as showed in Lines 5-8 of Listing 3.4. To mitigate these problems, we introduce the Layered Aspects idiom.

### 3.2.2 Layered Aspects

Now, we propose the Layered Aspects idiom to implement flexible binding time for features. Similarly to Pointcut Redefinition, the basic idea of Layered Aspects is to have the feature code in an abstract aspect and two concrete subaspects to implement static and dynamic binding time. For the static binding time, we compile an empty concrete subaspect which inherits the feature code from the abstract aspect and allows its instantiation, likewise the Pointcut Redefinition idiom. For the dynamic binding time, we compile another concrete subaspect that intercepts the execution of the pieces of advice that implement feature code defined in the abstract aspect to dynamically decide whether the feature code is executed. We implement this using the `adviceexecution` pointcut provided by AspectJ. In the following, we show this idiom's design and how it tries to address the problems of Edicts and Pointcut Redefinition.

#### Design

Layered Aspects is implemented using AspectJ aspects, as in Pointcut Redefinition. To modularize a given feature, we implement its related code in an abstract aspect. This aspect may contain advice, pointcuts and intertypes associated with the feature code.

Then, we implement static and dynamic binding time by creating two new concrete subaspects inheriting the abstract one, as illustrated in Figure 3.3. An empty concrete aspect (`StaticBinding`) is necessary in the compilation to allow the `AbstractAspect` instantiation when the feature binding occurs statically. For dynamic binding of features, we compile the `DynamicBinding` aspect, which has code for dealing with different kinds of advice defined in `AbstractAspect`. For `before` and `after` advice, we have the `adviceexecution` pointcut, which intercepts these advice and only proceeds their execution if the feature is activated. For `around` advice, this does not work because if the feature is deactivated, the base code overridden by the around advice would not be restored. So `Dynamic Binding` contains redefinitions of pointcuts that are related to `around` advice, similarly to Pointcut Redefinition. Thereby, we compile the application with the `DynamicBinding` aspect to provide dynamic binding time for features.

Moreover, the structure of Layered Aspects uses aspect inheritance because the

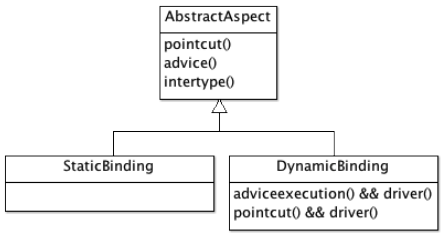



**Figure 3.3** The structure of Layered Aspects

`adviceexecution` pointcut works only for `before` and `after` advice. Dealing with `around` advice is complicated, because when the driver states the feature deactivation, we must restore the base code overridden by the `around` advice. Since there is no way to access the `proceed` join point of the advice intercepted by the `adviceexecution` pointcut, it is not possible to call it in a generic way. Thus, the pieces of `around` advice of the feature implementation must be deactivated one by one. Hence, we redefine the pointcuts related to `around` advice declarations and associate them with the driver in the `DynamicBinding` aspect. Thus, we apply the Pointcut Redefinition design for around advice.

The aforementioned structure of Layered Aspects avoids feature code cloning, tangling and scattering for the same reasons Pointcut Redefinition does. Layered Aspects does not duplicate advice with code between `StaticBinding` and `DynamicBinding`, since these pieces of advice are defined only in `AbstractAspect`. Additionally, it does not tangle driver and feature code because we implement the driver mechanism only in `DynamicBinding`, which does not contain feature code. Furthermore, the feature code is not scattered between the concrete subaspects because we implement it solely in `AbstractAspect`. However, Layered Aspects only increases its implementation size when several `around` advice are present due to the discussed `adviceexecution` pointcut. In the following, we provide more details about Layered Aspects.

### Example

To better explain Layered Aspects, consider the *Checksum* feature introduced in Section 3.1. We show how to apply flexible binding time for this feature and how this idiom addresses the Edicts shortcomings. We omit **Feature implementation** and **Implementing static binding time** explanation because it is identical to the one presented in Section 3.2.1. However, Section A.3 from Appendix A contains the Checksum complete implementation using this idiom. The difference of Layered Aspects consists of implementing the dynamic binding time, as follows.

**Implementing dynamic binding time.** Now, we show how Layered Aspects allows dynamic feature activation. Listing 3.5 shows how we implement dynamic binding of features. Line 3 defines the driver, which is a pointcut that checks if the checksum property corresponds to `true` or `false` in a property file, equally to the driver defined in line 3 of Listing 3.4. This way, the feature code is executed depending on this pointcut. Additionally, for dynamic feature binding, Lines 7-10 implement the `adviceexecution` pointcut to deal with `before` and `after` advice. If the `driver()` condition corresponds to `false`, the feature is deactivated, so this pointcut intercepts the pieces of advice defined in the `ChecksumAbstract` aspect, but it does not call the `proceed()` join point within the advice defined in Lines 7-10, so the feature code is not executed. On the other hand, if the `driver()` condition is `true`, the feature is activated, so this pointcut does not intercept any advice declaration, thus the feature code is executed. Moreover, line 5 redefines the `addPrevOffset` pointcut, which is defined in the `ChecksumAbstract` aspect, in order to associate it with the driver because this pointcut is related to an `around` advice. As explained in Section 3.2.2, such type of advice is handled separately, it is deactivated one by one. Therefore, we implement the dynamic binding following the Pointcut Redefinition structure for `around` advice.

Last but not least, returning `null` in Line 9 (Listing 3.5) is not harmful when the feature is deactivated because we apply the `adviceexecution` pointcut only for `before` and `after` advice, it does not intercept `around` advice. This way, when the feature is deactivated, the new pointcuts related to `around` advice are not applied and so the `adviceexecution()` does not intercept the execution of the advice defined in Lines 15-17 (Listing 3.3). If we remove the `null` statement, we may have a compilation error, since the `adviceexecution()` pointcut tries to intercept the execution of pieces of advice that return an `Object` or a primitive type.

**Listing 3.5** ChecksumDynamic.

```
1  aspect ChecksumDynamic extends ChecksumAbstract {
2   ...
3   pointcut driver() : if(new Driver().isActivated("checksum"));
4   ...
5   pointcut addPrevOffset() : ChecksumAbstract.addPrevOffset() && driver();
6   ...
7   Object around() : adviceexecution() && within(com.checksum.ChecksumAbstract)
8    && !driver() {
9     return null;
10   }
11 }
```

Despite reducing some of the issues, this idiom could scatter driver code when

several `around` advice are defined because we redefine the pointcuts related to it, as discussed above. The Pointcut Redefinition idiom presents the same deficiency, however it redefines all the pointcuts whereas Layered Aspects redefines only the pointcuts related to `around` advice. Hence, Layered Aspects design does not reduce the implementation size comparing to Pointcut Redefinition in such cases. To handle these issues, we present next the Flexible Deployment idiom.

### 3.2.3 Flexible Deployment

The Flexible Deployment idiom uses dynamic deployment of aspects provided by CaesarJ [AGMO06] to allow feature binding time flexibility. CaesarJ is an aspect-oriented language that extends Java and supports flexible deployment of classes and advanced object-oriented modularization mechanisms. The idea is to define a CaesarJ class, which can define aspect-oriented constructs, as explained in Section 2.2.2, to modularize the feature code and two additional CaesarJ classes to implement static and dynamic binding. For static binding, we define a statically deployed CaesarJ class that inherits the other CaesarJ class that contains the feature code, as showed in Listing 2.6 of Section 2.2.2. For dynamic binding, a deployed CaesarJ class implements the driver mechanism. It activates the feature by dynamically deploying the CaesarJ class that contains the feature code and consequently allowing its execution, as illustrated in Listing 2.7 of Section 2.2.2. We provide more details about this idiom in the following sections.

#### Design

Flexible Deployment is implemented using CaesarJ classes. We define a CaesarJ class to modularize a given feature. It may contain pointcuts, advice, and wrapper classes associated with the feature implementation. These wrapper classes contain feature code and they wrap a class from the base code, as explained in Section 2.2.2. Unlike AspectJ intertype declarations, these wrapper classes do not introduce feature code in base code classes. Figure 3.4 shows the structure of Flexible Deployment.

Furthermore, we implement static binding by defining an empty deployed CaesarJ class (`StaticBinding`) that inherits from `FeatureCaesarJClass`. Thereby, when both CaesarJ classes are present in a build, the feature is statically activated. For dynamic binding, we define a separate CaesarJ class (`DynamicBinding`), which contains the driver mechanism implemented as a pointcut and an advice. We provide more details in this section.

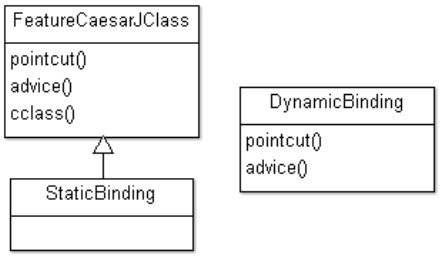



**Figure 3.4** The structure of Flexible Deployment

This idiom does not clone, tangle, or scatter code. The feature code is implemented only in a separate CaesarJ class. Therefore, there is no cloning of pieces of advice and there is no feature code scattering throughout the classes either. Since we implement the driver mechanism in a separate CaesarJ class, the driver code is not tangled with feature code. Moreover, the idiom implementation size does not increase when several pointcuts are present as Pointcut Redefinition does.

**Listing 3.6** ChecksumCaesarJClass.

```
1  cclass ChecksumCaesarJClass {
2   ...
3   pointcut readHeader() : call(void EntryHeader.readHeader());
4   ...
5   pointcut addPrevOffset(int entrySize)
6    : execution(ByteBuffer LogManager.addPrevOffset()) && args(entrySize);
7   ...
8   after() throws DatabaseException : readHeader() {
9    if (doChecksumOnRead) {
10    validator = new ChecksumValidator();
11    ...
12   }
13  }
14  ...
15  ByteBuffer around(int entrySize) : addPrevOffset(entrySize) {
16   ...
17   return proceed(entrySize);
18  }
19  ...
20  cclass FileReaderCaesarJ wraps FileReader {
21  ...
22   void validateChecksum() {
23    ...
24   }
25  ...
26  }
27 }
```

### Example

Now, we use the *Checksum* feature to describe how to implement flexible binding time using the Flexible Deployment idiom more clearly. In addition, Section A.4 contains the Checksum complete implementation using this idiom.

**Feature implementation.** The CaesarJ class that contains feature code may define pointcuts, advice, and wrapper classes, as illustrated in Listing 3.6. Differently from the other two AspectJ-based idioms, CaesarJ does not support intertypes, so we define wrapper classes (Lines 20-26) in `ChecksumCaesarJClass` instead. The `FileReaderCaesarJ` wrapper class is a dynamic extension of the `FileReader` class. It can introduce new state and operations [AGMO06]. For example, we introduce the `validateChecksum` method related to the feature implementation (lines 22-24). The `ChecksumCaesarJClass` encompass the feature implementation in this example. Lines 3-6 define pointcuts that match certain join points in the base code where we should introduce feature code. Lines 8-18 define pieces of advice that contains this feature code.

**Implementing static binding time.** We define a **deployed** CaesarJ subclass extending `ChecksumCaesarJClass`, as illustrated in Listing 3.7. Since instantiation does not automatically activate a class in CaesarJ, we have to declare the CaesarJ subclass with the deployed keyword (Line 1). Therefore, we activate the feature execution by including both CaesarJ classes in the project build. On the other hand, to deactivate feature execution, we do not include any of these CaesarJ classes.

**Listing 3.7** ChecksumStatic.

```
1 deployed cclass ChecksumStatic extends ChecksumCaesarJClass {
2 }
```

**Listing 3.8** ChecksumDynamic.

```
1 deployed cclass ChecksumDynamic {
2   pointcut pc_jarmain() : execution(* JarMain.main(..));
3   before() : pc_jarmain() {
4     if (new Driver().isActivated("checksum")) {
5       deploy new ChecksumCaesarJClass();
6     }
7   }
8 }
```

**Implementing dynamic binding time.** Now, we describe how the Flexible Deployment idiom allows dynamic feature activation. Listing 3.8 shows the driver mechanism implementation in a **deployed** CaesarJ class. In this case, we define a pointcut

that intercepts the system main method execution in Line 2. This allows the advice defined in Line 3 to dynamically deploy the CaesarJ class that contains feature code (`ChecksumCaesarJClass`) before the main method execution. Thereby, the feature code is executed depending on the driver mechanism. Furthermore, this driver mechanism is implemented according to the application requirements. Thus, it is not necessarily implemented as in Listing 3.8. As we mentioned in Section 3.2.1, it could vary from simple GUIs to complex sensors.

Indeed, Flexible Deployment implementation does not clone and scatter feature code because we do not need to duplicate it among the classes that implement static and dynamic binding time, so we implement feature code only in one class. In addition, this idiom does not tangle feature and driver code since we define a separate class to implement the driver mechanism. Finally, Flexible Deployment reduces the implementation size comparing to the other idioms because its size does not vary due to certain advice, as in Layered Aspects. However, we cannot apply the Flexible Deployment idiom for some features because CaesarJ does not support all AspectJ constructs. Therefore, if a given feature implementation uses these constructs, we would not be able to provide flexible binding time with Flexible Deployment. We provide further details in Section 4.2.7.

## 3.3 Feature interaction

We already explained how to implement flexible binding time for features in the previous sections. However, it might be the case that part of the code of a feature alters the behavior of another feature. In this case, we use separate aspects to modularize code related to both features at the same time, and we implement flexible binding time to such separate set of aspects. Figure 3.5 illustrates this scenario. Part of the code of Feature A alters the execution of Feature B. Hence, we modularize the joint selection of Features A and B in an aspect. Thus, we implement flexible binding time for this aspect. The main difference to the approach we presented in the previous sections is that we implement flexible binding time for the interaction between features instead of a single feature.

For example, Figure 2.2 illustrates a product line that define applications for `Desktop` or `Mobile` environments. Thus, these two features are modularized in aspects but their binding is only static since it does not make sense to alter the environment dynamically. Thereby, despite of they are modularized in aspects, we do not implement our idioms in such case. However, notice that the `NextPiece` and `Record` features may be activated dynamically. But, when we identify their code, we observe they contain code tangled

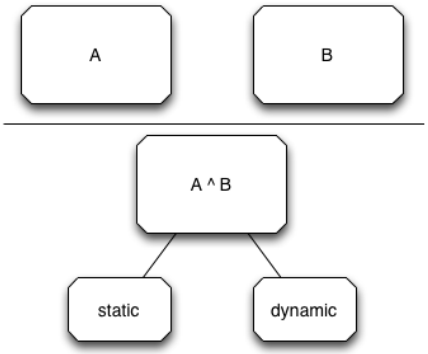
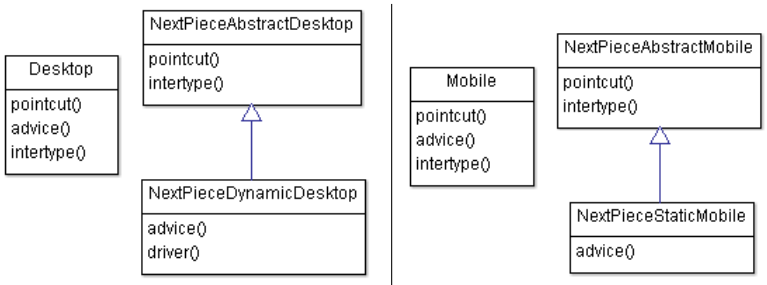



**Figure 3.5** Feature Interaction

with the `Desktop` or `Mobile` aspects.




(a) Tetris for desktop. (b) Tetris for mobile.

**Figure 3.6** Tetris SPL.

To deal with such situation, we implement two products (`NextPiece` and `Record`) for desktop and mobile environment. For instance, Figures 3.6(a) and 3.6(b) illustrate the structure for this case using the Edicts idiom. Notice that Figure 3.6(a) contains `NextPiece` and `Desktop` code. Hence, the `Desktop` aspect modularizes the code exclusively associated to the desktop environment, it may define pointcuts, advice, and intertype declarations. The `NextPieceAbstractDesktop` aspect modularizes the NextPiece code and the `NextPieceDynamicDesktop` aspect implements dynamic binding time following the structure of Edicts. Analogously, we implement the mobile product, as illustrated in Figure 3.6(b).

Last but not least, we only provide dynamic binding for the desktop environment and static binding for the mobile environment for this case study. Although, we could have both static and dynamic binding for each environment in other scenario. We give further explanation in Section 4.1.1.

# 4 Evaluation

To evaluate the idioms discussed in the previous chapter, in this chapter we discuss empirical assessments we performed regarding code cloning, scattering, tangling, and size. In particular, we implemented the four idioms in 18 features of four applications. We now present the study settings detailing the case studies and their features, which are implemented using the flexible binding time idioms, in Section 4.1.1. In addition, we discuss the GQM approach used to assess the idioms in Section 4.1.2. In Section 4.1.3, we explain the assessment procedures we followed to do this work. Moreover, we observed modularity problems in the Edicts idiom implementation. This way, we discuss the evaluation regarding modularity and code quality metrics, such as code cloning, scattering, tangling, and size in Section 4.2. Therefore, we assess the implementations of the idioms to confirm that we reduce the modularity problems that we identified in Edicts, for our idioms. In addition, to confirm that there is no difference in the execution of a given feature code implemented by different idioms, we evaluate if the execution behavior changes between these idioms, that is, we use the SafeRefactor tool to compare the behavior of the same feature implemented with two different idioms. We conclude this chapter presenting some threats to the validity of our studies. At last, we discuss the advantages and disadvantages of our idioms.

## 4.1 Study settings

To evaluate the idioms we discussed in Chapter 3, we performed an empirical assessment of the idioms focusing on software modularity. Now we detail the study configuration, which involved the implementation of flexible binding time using Edicts, Pointcut Redefinition, Layered Aspects, and Flexible Deployment in 18 features of four case studies. Firstly, we explain the four case studies we selected to this work, and then we discuss the

Goal-Question-Metric (GQM) approach [BCR94] that was used to evaluate our work. To guide our evaluation, we present the goals we aim to reach, which consists of assessing idioms to implement flexible binding times for features regarding software modularity. In addition, we present the research questions we intend to investigate, such as what idiom helps to reduce code cloning. Further, this section outlines the metrics used to measure software modularity and consequently answer these questions. Finally, to explain how we performed this work, we explain the assessment procedures.

### 4.1.1 Case studies

This section presents the applications and their features that require flexible binding time support. We apply the three idioms introduced in Section 3.2 plus Edicts.

BerkeleyDB, which is one of our case studies, already was a product line [KAB07]. Therefore, we reviewed and refactored the ten features implementations we selected from BerkeleyDB to comply with the way we implement the other eight feature implementations we selected from the other three case studies. One example of this refactoring consists of separating pointcuts from advice declarations. Leaving them together would alter the evaluation and introduce bias, such as increasing code cloning for the Edicts idiom because following its structure, presented in Section 3.1, we would have to duplicate pointcuts and advice instead of only pieces of advice among the concrete subaspects. Besides BerkeleyDB, we considered three other applications: Tetris [Tet09], Freemind [Fre09], and ArgoUML [Arg09]. They were plain object-oriented applications. We modularized some of their features with aspects obtaining an SPL before applying the idioms. In what follows, we provide more details about these four applications.

#### Tetris

Tetris [Tet09] is an open-source implementation in Java of the well know Tetris game. It runs on the Java ME (Micro Edition) platform. We created a product line by modularizing the code related to Java ME. Thus, we are able to code and add Java SE (Standard Edition) support as well. This way, we can run Tetris for both platforms. Figure 2.2 in Section 2.1.1 shows the feature model of our Tetris SPL. Note that *Mobile* and *Desktop* are alternative features that represent the platforms we can run this case study. We implement flexible binding time for *Record* and *NextPiece* optional features. The red circles in Figure 4.1 illustrate both features. The *Record* feature shows to the user what is the highest score any player has already achieved. The second feature, named *NextPiece*, shows the next

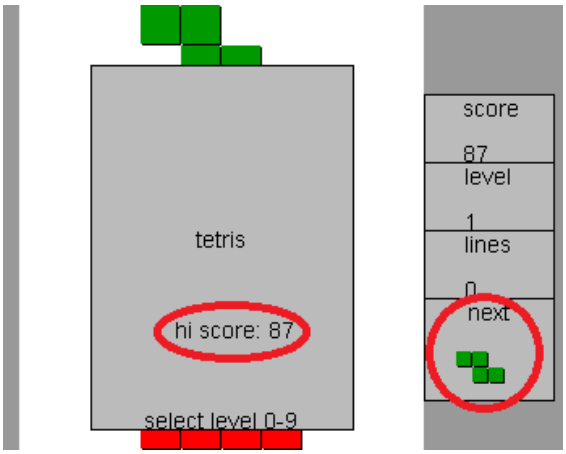



**Figure 4.1** Tetris Features

piece which is about to drop on the screen. This product line has about 1500 lines of code. The *Record* and *NextPiece* features have nearly 400 lines of code within aspects and classes. We chose these features because they represent a scenario where both optional features interact with alternative features, as we explained in Section 3.3.

For the Java SE platform, we provide a dialog box in the beginning of the execution to let the user choose what features should be activated. Thus, we provide dynamic feature binding. On the other hand, for the Java ME platform, it is desirable to avoid overhead introduced by dynamic binding [RSSA08] due to restrictions of performance, so we provide a simple user interface without this dialog box, so we statically activate or deactivate the features.

Therefore, it is possible to generate different products: (i) features dynamically bound for the Java SE platform and (ii) features statically bound or (iii) unbound for the Java ME platform. Notice that the *Record* and *NextPiece* features do not interact. Therefore, we may have any combination. Both may be activated or deactivated or one may be deactivated while the other is activated or vice-versa.

### Freemind

Freemind [Fre09] is an open-source system used to construct diagrams to organize ideas by using mind maps. It is written in Java and runs on the Java SE platform. We selected this case study because it is a widely used application, which presents code related to particular functionalities scattered throughout several layers and classes. In this context, we modularize two features to create a product line and provide flexible binding time. Figure 4.2 illustrates the feature model of our resulting SPL.

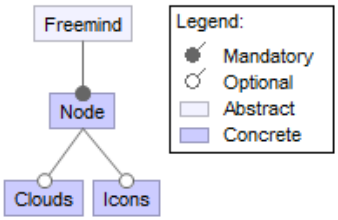



**Figure 4.2** Freemind SPL feature model.

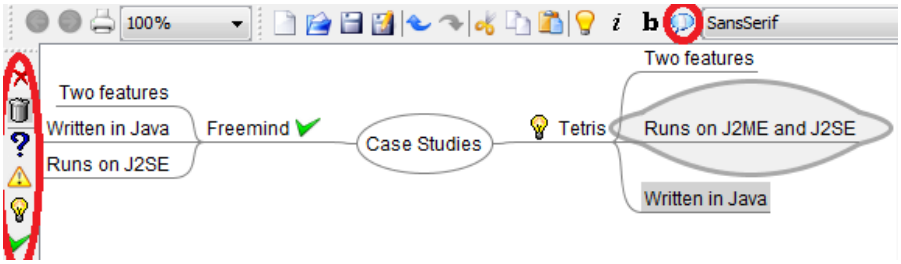



**Figure 4.3** Mind map constructed in Freemind.

The *Clouds* optional feature (right hand-side circle) may alert an important node from the mind map. For instance, in Figure 4.3, we use a cloud to call our attention about what platforms Tetris runs on. The *Icons* optional feature decorates these nodes. The Tetris and Freemind nodes contain icons. We chose these features due to their code representativeness. They are crosscutting, scattered and tangled throughout different architecture layers. We also use several different AspectJ constructs to modularize these features. Figure 4.3 illustrates a mind map in Freemind that organizes the information of our case studies. The circles represent the two features we modularized.

The Freemind product line has about 67000 lines of code and both features have approximately 4000 lines. Note that following the Figure 4.2, we can have any combination of optional features. In this context, we can generate different products with dynamic and static feature **bind** and static feature **unbind**.

### ArgoUML

ArgoUML [Arg09] is an open-source UML modeling tool written in Java that includes support for standard UML 1.4 diagrams. We selected this case study because the application code is separated into subsystems that have different responsibilities and are organized in layers. This way, the feature code should be scattered within a subsystem instead of the whole application as in Freemind, which turns its feature code different from the other case studies increasing the representativeness of these features. In this

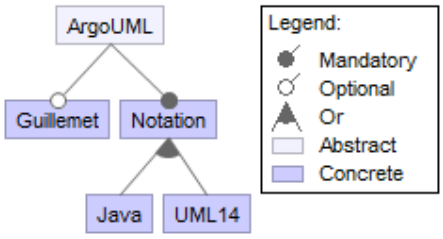



**Figure 4.4** ArgoUML SPL feature model.

context, we create a product line by modularizing two features and introducing flexible binding time for both.

For the first feature, we focus on the Notation subsystem which defines the notation language used in UML diagrams. ArgoUML provides two notations: *UML 1.4* and *Java*. This is an OR feature, so we may have only *UML 1.4*, only *Java* or both. We also consider the *Guillemets* optional feature. It is responsible for showing the symbols "<<" ">>" to accommodate the stereotypes of classes in the diagrams. Despite the simplicity, the *Guillemets* feature code is scattered throughout many modules of ArgoUML.

The ArgoUML product line has nearly 113000 lines of code and 468 of feature code. Figure 4.4 shows the feature model of this product line. Note that we may have any combination between the two selected features. This way, we can generate different products with dynamic and static feature **bind** and static feature **unbind**, equally to the Freemind case study.

### BerkeleyDB

BerkeleyDB[1] is an open-source database written entirely in Java. It uses the Java Environment advantages to simplify development and redeployment and provides simple store key/value pairs of arbitrary data. For this work, we use the BerkeleyDB product line [KAB07]. Kästner et al. modularized several features using aspects.

Table 4.1 explains the responsibility of each feature. The *IO* and *NIO* features are alternatives, which means that only one may be present at a time. The other features are optional. However, these optional features have interactions between each other. For example, *Truncate* depends on *Delete* to delete the database before creating a new empty one. In other words, *Delete* must be present in the product when *Truncate* is activated. We discuss these interactions later in Section 4.2.7. Due to the quantity of features, we

[1] http://www.oracle.com/technetwork/database/berkeleydb/overview/index-093405.html

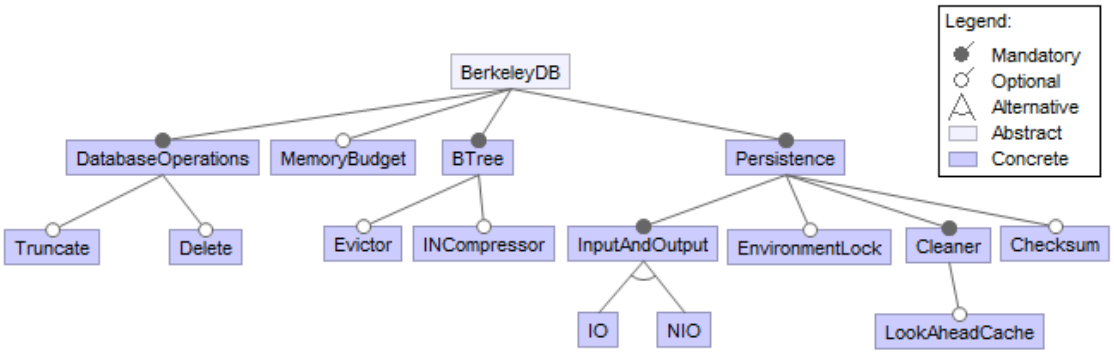



**Figure 4.5** BerkeleyDB SPL feature model.

**Table 4.1** BerkeleyDB Features

| Feature | Definition |
|---|---|
| Checksum | Checksum read and write validation of persistence subsystem. |
| Delete | Deletes database. |
| EnvironmentLock | Prevents two instances on the same database directory. |
| Evictor | Reduces memory consumption by evicting non persistent nodes from the database tree. |
| INCompressor | Removes delete entries and empty nodes from the tree. |
| IO | Classic I/O implementation. |
| Look Ahead Cache | Keeps track of memory used, and when full (over budget), the node offsets should be queried and removed. |
| Memory Budget | Calculates the available memory for the database and how to apportion it between cache and log buffers. |
| NIO | New I/O implementation. |
| Truncate | Deletes the database and creates a new one without the previous data. |

can have several number of product configurations.

This product line, has approximately 32000 lines of code. All the ten features selected sum up approximately 4000 lines of code. Figure 4.5 illustrates the feature model of this case study. Differently from the other case studies, we cannot have any combinations of features. As aforementioned, some implementation of features depend on other implementation of features to execute correctly. This limits the possible products we may generate.

### 4.1.2 GQM

We use the Goal-Question-Metric (GQM) approach to drive the evaluation process. We first specify the goal (Section 4.1.2), which states what we aim to evaluate. Second, we define a set of questions to characterize the way we perform the assessment (Section 4.1.2). Finally, we refine the questions into metrics in order to obtain answers in a quantitative way (Section 4.1.2) [BCR94].

### Goal

The main goal of our evaluation consists of assessing idioms to implement flexible binding times for features. Notice that such assessment regards software modularity, so we are concerned about aspects like (i) code duplication, (ii) feature and driver code scattering, (iii) tangling between driver and feature code, and (iv) the source code size.

### Questions

In what follows, we detail the questions we investigate in this paper related to the metrics we use to answer them.

Which idiom contributes to reduce:

**Q1-** the code duplication when implementing binding time flexibility?

1. Pairs of Cloned Code (PCC)

**Q2-** the driver and feature code scattering?

1. Degree of Scattering across Components (DOSC)
2. Degree of Scattering across Operations (DOSO)
3. Concern Diffusion over Components (CDC)

**Q3-** the tangling between the driver and feature code?

1. Degree of Tangling within Components (DOTC)
2. Degree of Tangling within Operations (DOTO)

**Q4-** the lines of code and number of components?

1. Source Lines of Code (SLOC)
2. Vocabulary Size (VS)

### Metrics

In order to answer the questions just presented, we evaluate the implementation of the idioms in the case studies using a metrics suite. Most metrics we chose have already been defined and successfully used to measure quality factors in several works [EZS+08,

BB09, GSF+06, Ape07, FGS+05, BBM96]. All of these metrics are defined to be used at the implementation level, which we focus on this work.

**Clone**

- **Pairs of Cloned Code (PCC)** - It measures the number of pairs of duplicated code based on tokens.

**Modularity**

- **Degree of Scattering across Components (DOSC)** - It measures how distributed is a concern code across components (classes or aspects). It varies from 0 to 1. If DOSC is 0, then the code of a concern is in a single component. On the other hand, if DOSC is 1, then the code of a concern is equally divided among all considered components [EAM07];
- **Degree of Scattering across Operations (DOSO)** - Similarly to DOSC, DOSO measures how distributed is a concern across methods and advice. It varies from 0 to 1. If DOSO is 0, then the code of a concern is in a single method or advice. On the other hand, if DOSO is 1, then the code of a concern is equally divided among all considered methods and advice [EAM07];
- **Concern Diffusion over Components (CDC)** - Number of components that has code of a concern [GSF+05];
- **Degree of Tangling within Components (DOTC)** - It measures how dedicated a component (class or aspect) is to one or more concerns under consideration. Like DOSC and DOSO, it varies from 0 to 1. If DOTC is 0, then the code of a component is totally dedicated to one concern. On the other hand, it is 1 if the code of a component is dedicated to all concerns under consideration [Ead08];
- **Degree of Tangling within Methods (DOTO)** - It measures how dedicated a method or advice is to one or more concerns under consideration. It varies from 0 to 1. If DOTO is 0, then the code of a method or advice is totally dedicated to one concern. On the other hand, it is 1 if the code of a method or advice is dedicated to all concerns under consideration [Ead08].

**Size**

- **Source Lines of Code (SLOC)** - Number of source lines of a component (e.g., classes or aspects);

- **Vocabulary Size (VS)** - Number of program components (e.g., classes or aspects);

We use Pairs of Cloned Code (PCC) in Section 4.2.1 to answer Question 1, as it may indicate a design that could increase maintenance costs [BYM+98] because a change would have to be replicated to the duplicated code as well. To answer Question 2, we use CDC, DOSC, and DOSO in Section 4.2.2 to measure the implementation scattering for each idiom. This way, we measure code scattering through different perspectives, such as number of components in which the code is scattered, and how dedicated the code is with respect to components or methods. To answer Question 3, we measure the tangling between driver and feature code considering the DOTC and DOTO metrics in Section 4.2.3. Hence, our evaluation considers code tangling within components and methods. In addition, Source Lines of Code (SLOC) and Vocabulary Size (VS) are well known metrics for quantifying a module size and complexity. So, in Section 4.2.4, we answer Question 4 measuring the size of each idiom in terms of lines of code and number of components.

### 4.1.3 Assessment procedures

To compute the metrics values and perform our assessment, we follow the following procedure. We detail these procedures in five steps.

**1- Selection of case studies.** In order to generalize the results of our study to other contexts, we selected four different applications and eighteen different features. The applications we chose represent different sizes, purposes, and architectures. Additionally, the selected features have different sizes, architectures, types, granularity, and complexity. Moreover, they have different types, such as optional, alternative, OR, and mandatory [KCH+90].

**2- Feature code identification and assignment.** After choosing the feature, we apply Prune Dependency [EAM07] rules to identify feature code scattered throughout the application. These rule state that "*a program element is relevant to a concern if it should be removed, or otherwise altered, when the concern is pruned from the application*". By following these rules, two different people can identify the same code related to a given concern. We chose this rule to reduce introducing bias while identifying feature code. We execute this task manually by following the Prune Dependency rules. We use comments to assign the program elements related to each feature. This way, two different researchers could identify and assign our feature code.

**3- Code modularization.** For this phase, we extracted the feature code assigned from

the classes to aspects, except for BerkeleyDB, which the feature code was modularized before. This way, we aim at separating the feature and core code. This allows the creation of product lines from the applications. After this phase, the application should be independent from the modularized feature. For example, the application may compile and execute properly without the optional feature code.

**4- Flexible binding time implementation.** We then implement the idiom for the modularized features that we wish to provide binding time flexibility. We use the chosen idiom to control whether the feature code is executed. In this work, we apply the four idioms for each feature, but only one idiom is necessary to obtain binding time flexibility.

**5- Evaluate idioms.** In this phase, we use the GQM approach to evaluate the four idioms. Our goal is to assess the implementation of the idioms in the case studies. We elaborate four questions with respect to points that we want to investigate about the idioms. Then, we answer the questions by analyzing the measures obtained for the metrics we selected. In addition, we investigate possible changes concerning the behavior of the implementations.

## 4.2 Results

In this section, we evaluate the presented idioms. We aim at answering the questions outlined in Section 4.1.2, so that we answer each question in the following subsections. In some sections, we only show the metric results for some features, which are enough to drive our discussion. However, in Appendix B, we show the complete metric results. Moreover, the material produced in this work is available in our website[2]. In addition, we do not present the Flexible Deployment idiom results for the Tetris product line because CaesarJ does not support some aspect constructs needed to modularize the features of Tetris, we provide more details in Section 4.2.7.

### 4.2.1 Cloning

To answer Question 1 and try to determine which idiom reduces code cloning, we use the CCFinder [ccf07] tool to obtain the *PCC* metric results. CCFinder is a widely used tool to detect cloned code, and several works used it with the same purpose [SR05, KG06, BvDvET05, AEKHG05, DHJ+08]. We use 40 as the minimum clone length (in tokens), which means that to be considered cloned, two pairs of code must have at least 40 equal

[2]http://twiki.cin.ufpe.br/twiki/bin/view/SPG/FlexibleBindingTime

**Table 4.2** PCC metric results

| Feature | Edicts | PointcutRedef | Layered Aspects | FlexDeploy |
|---|---|---|---|---|
| **Icons** | 19 | 4 | 5 | 4 |
| **Clouds** | 5 | 1 | 1 | 1 |
| **Notation** | 9 | 9 | 9 | 8 |
| **Guillemets** | 4 | 0 | 0 | 0 |
| **NextPiece-Desktop** | 0 | 0 | 0 | - |
| **NextPiece-Mobile** | 0 | 0 | 0 | - |
| **Record-Desktop** | 0 | 0 | 0 | - |
| **Record-Mobile** | 0 | 0 | 0 | - |
| **EnvironmentLock** | 2 | 0 | 0 | 3 |
| **Checksum** | 5 | 0 | 0 | 0 |
| **Truncate** | 2 | 2 | 2 | 5 |
| **Delete** | 9 | 3 | 2 | 0 |
| **LookAheadCache** | 2 | 0 | 0 | 0 |
| **Evictor** | 0 | 0 | 0 | 0 |
| **NIO** | 0 | 0 | 0 | 0 |
| **MemoryBudget** | 34 | 17 | 7 | 1 |
| **IO** | 0 | 0 | 0 | 0 |
| **INCompressor** | 2 | 2 | 1 | 1 |

tokens. Pairs of similar code that have 39 tokens are not considered cloned, so that they are discarded by CCFinder. Results using less than 40 similar tokens are discarded due to the frequent appearance of undesired clones, such as package names. On the other hand, we do not apply a higher minimum clone length because we miss some relevant pairs of cloned code. We chose to use a token-based detection of code cloning because representing a source code as a token sequence enables to detect clones with different line structures, which cannot be detected by line-by-line algorithm [KKI02]. This allow us to detect more clones in our case studies. In addition, we use 12 as the token set size (TKS). This way, the TKS is not small enough to consider that the cloned code fragment is a simple statement, such as a series of variable declarations, which is often present in classes and in most of the cases is irrelevant. On the other hand, the TKS is not higher because we may miss some interesting duplication of code because the cloned code fragment would be very large. Table 4.2 summarizes the results of the Pairs of Clone Code (PCC) metric.

As in Section 3.1, Edicts duplicates feature code in the concrete subaspects. This leads to high *PCC* rates specially for large features that define many pieces of advice, such as the *Memory Budget* and *Icons* features. The *NextPiece* and *Record* features do not present clones because they have only static binding time for the mobile version, and only dynamic binding for the desktop version. Therefore, Edicts do not clone feature code in the concrete subaspects, since it has only one subaspect to implement static or

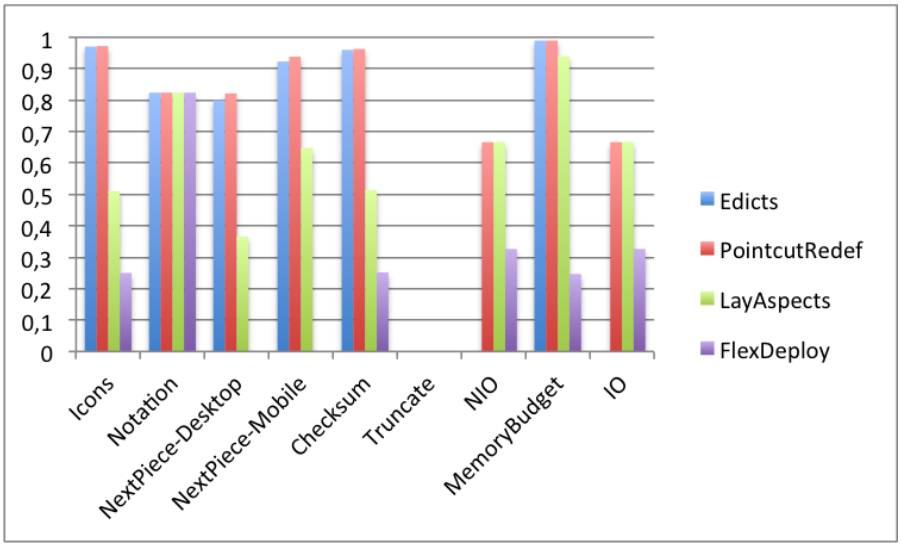



**Figure 4.6** DOSO Driver metric results

dynamic binding time. The Pointcut Redefinition idiom presents high *PCC* rate for the *Memory Budget* feature because it is a large feature, which contains many pointcuts. Hence, following the Pointcut Redefinition idiom design, we redefine these pointcuts in the concrete subaspects, and consequently we clone their signatures. Layered Aspects and Flexible Deployment have low *PCC* results for most features.

### 4.2.2 Scattering

As mentioned in Section 4.1.2, we use *DOSO*, *DOSC*, and *CDC* to analyze feature and driver code scattering for each idiom. Feature and driver are different concerns, so we analyze them separately. By using these metrics, we intend to answer Question 2 from Section 4.1.2. In particular, we study the driver code scattering and the feature code scattering in the following.

#### Driver

Now, we discuss the driver code scattering. Figure 4.6 presents the results considering the *DOSO* metric. Edicts and Pointcut Redefinition have the worst results because they scatter driver code throughout many program elements. For example, the Edicts idiom introduces driver code into several pieces of advice. Moreover, Pointcut Redefinition introduces driver code into the redefined pointcuts in the concrete subaspect. On the other hand, Layered Aspects only increases the *DOSO* results in cases which we redefine the pointcuts related to around advice (see Section 3.2.2). Accordingly, Flexible Deployment

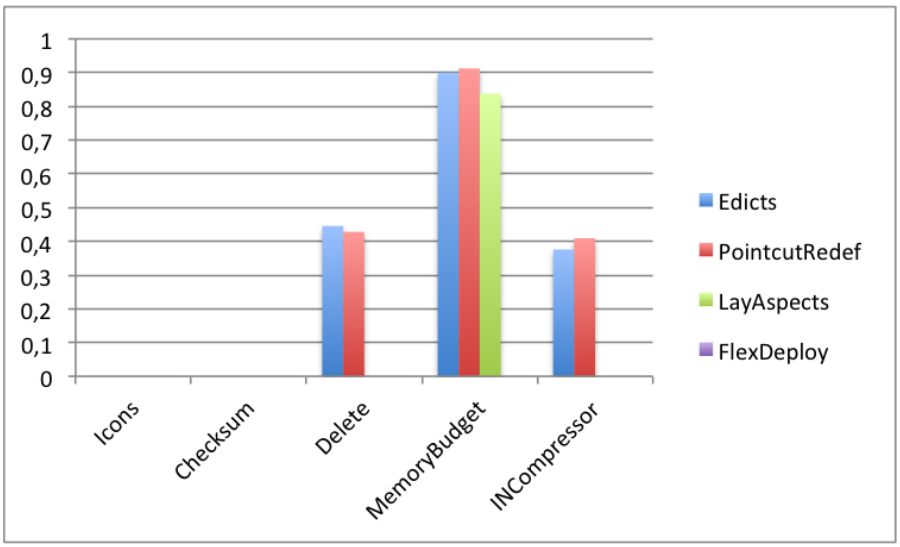



**Figure 4.7** DOSC Driver metric results

has low driver scattering due to its dynamic deployment of aspects which implements the driver mechanism in only one pointcut and advice. The *Truncate*feature has zero *DOSO* because it does not implement pieces of advice. The *Notation* feature has equal results for all the idioms because its implementation is similar independently of the idiom. This happens because the *Notation* feature consists of a subsystem in the architecture of ArgoUML. Hence, this feature is not scattered throughout the source code so that we only need to deactivate this subsystem. The *IO* and *NIO* features have only one advice, so the *DOSO* is zero for them, since there is only one `if` statement in a single advice.

To provide the scattering results in another perspective, we also consider the *DOSC* metric. This allow us to identify in which cases the code scattering occurs in a class or aspect level. Although driver code scattering is common when considering pointcuts and advice, it is unusual when considering classes or aspects because most of our features do not need more than one aspect in its implementation. Therefore, we implement the driver code only in one concrete subaspect. Figure 4.7 shows the results regarding the *DOSC* metric. The *Delete*, *Memory Budget*, and *INCompressor* features use more than one aspect to implement the feature code. Therefore, their *DOSC* results may vary because we implement the driver in more than one aspect too, since we have to implement static and dynamic binding time in concrete subaspects defined for each abstract aspect. This is specially bad for Edicts and Pointcut Redefinition since we may have to introduce driver code in the concrete subaspects to implement dynamic binding time. On the other hand, for the other features, the *DOSC* is zero because the driver code is implemented only

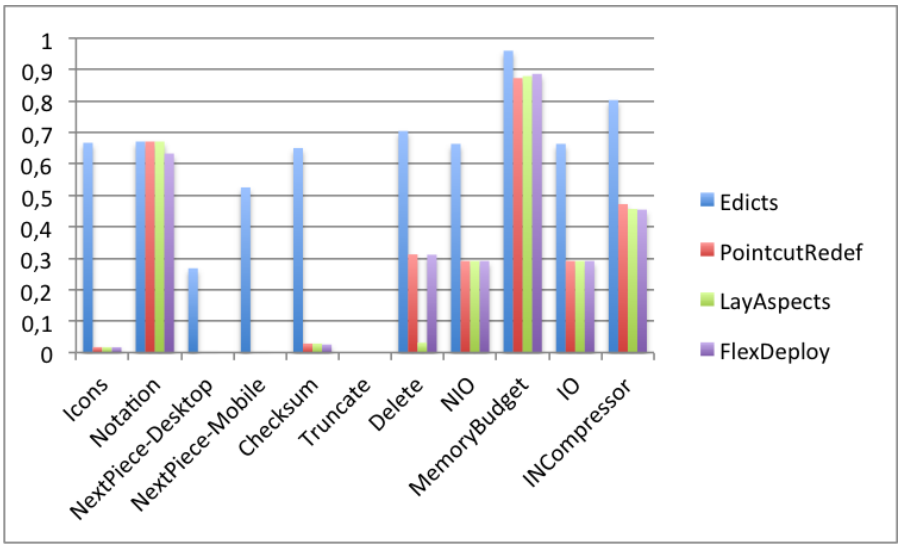



**Figure 4.8** DOSC Feature metric results

in one aspect. Therefore, it is not scattered throughout the components that implement feature code.

### Feature

Now, we discuss the feature code scattering. The *DOSC* metric depends on the number of components related to the feature implementation. Therefore, we consider two perspectives. From a package perspective, the feature code is well modularized as its implementation is in a single package. This way, there is no scattering at the package level. On the other hand, one feature implementation may use multiple aspects, leading to a higher feature scattering.

According to the results showed in Figure 4.8, Edicts scatters feature code because its design leads to the implementation of feature code in the abstract aspect and also in the concrete subaspects. In contrast, the other idioms only scatter feature code in cases multiple aspects are used in its implementation, as in *Memory Budget*. In these features, we used more aspects to modularize their code because they are large so each aspect would be responsible for a particular feature concern. In other cases, Layered Aspects, Pointcut Redefinition, and Flexible Deployment present low *DOSC* results. This is expected due to the concrete empty aspect used to allow the feature code instantiation, as explained in Section 3.2.

In addition, we apply another metric to measure the scattering through a different perspective. The *CDC* metric allows us to identify potential differences among the idioms

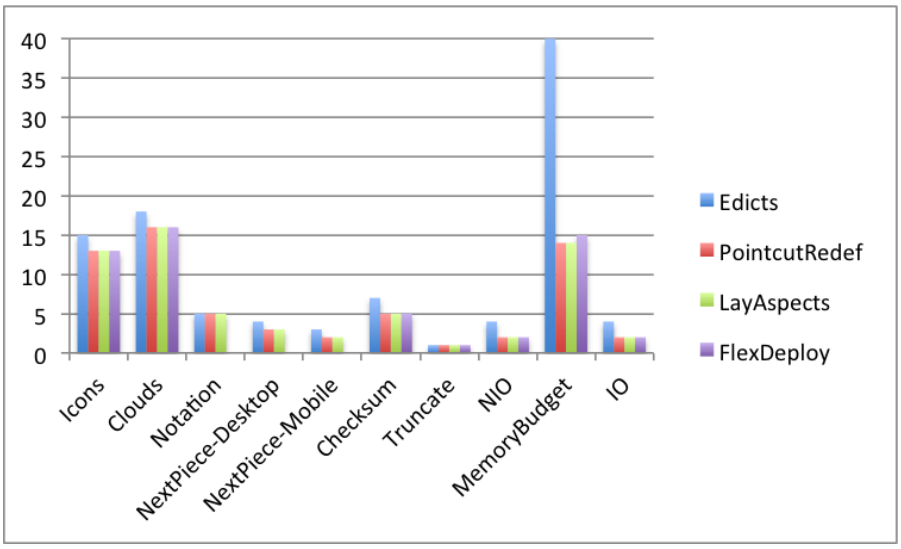



**Figure 4.9** CDC metric results

(Figure 4.9).

The difference between the idioms appears when we use more than one aspect to modularize the feature code. Notice that Edicts and Pointcut Redefinition have the worst results due to their design. These idioms define two concrete subaspects to implement static and dynamic binding time for each abstract aspect. On the other hand, Layered Aspects and Flexible Deployment implementation contain less components because they do not have to define these concrete subaspects for each abstract aspect.

To sum up, *DOSC* and *CDC* show that our idioms reduce feature code scattering comparing to the Edicts idiom. The *CDC* metric complements *DOSC* because it shows that our idioms reduce feature code scattering also when we use multiple aspects to implement the feature, as in the *Memory Budget* feature.

### 4.2.3 Tangling

This section answers Question 3 by investigating the extent of tangling between feature and driver code. According to the principle of separation of concerns [Par72], one should be able to implement and reason about each concern independently.

In this work, we assume that *the greater is the tangling between feature code and its driver code, the worse is the separation of those concerns*. As mentioned in Section 4.1.2, we measure the Degree of Tangling within Methods (*DOTO*) and the Degree of Tangling within Components (*DOTC*). Figure 4.10 and 4.11 show the *DOTO* and *DOTC* results, respectively.

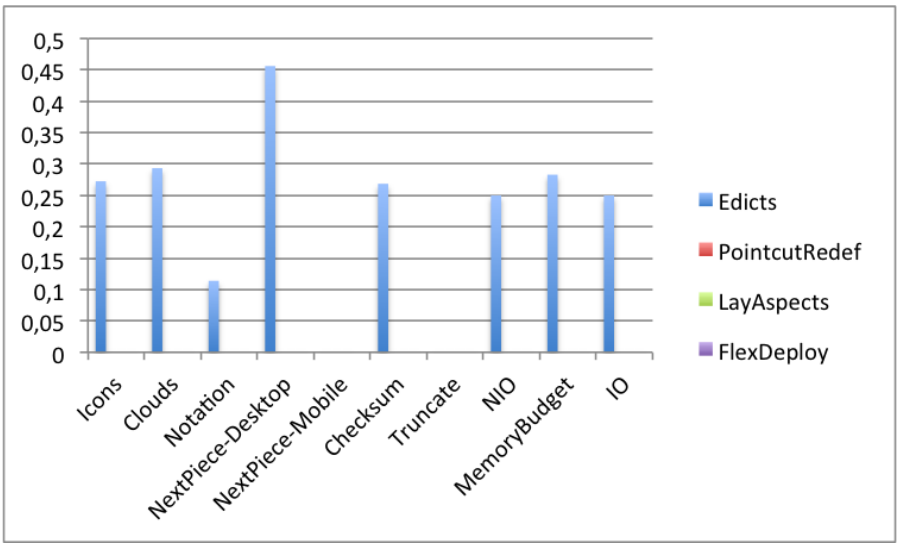



**Figure 4.10** DOTO metric results

Edicts has high *DOTO* rates because we implement driver code in each piece of advice. As a consequence, feature and driver code are tangled in the concrete subaspect that implement dynamic binding time. The other three idioms do not tangle driver and feature code since the abstract aspect contains only feature code and the concrete subaspect that implement dynamic binding contains only the driver implementation. Hence, there is no advice or pointcut with feature code defined in the Pointcut Redefinition, Layered Aspects, or Flexible Deployment implementation that is associated with driver code.

The *NextPiece-Desktop* feature presents the highest *DOTO* because its implementation contains only advice and following the Edicts design, all of these advice are tangled with driver code. As we explained in Chapter 3, the driver code is equally distributed among the pieces of advice, that is, they contain the same `if` statement, so it is the same amount of driver code that each piece of advice has. On the contrary, features that present low *DOTO* have few pieces of advice that contain feature and driver code comparing to the total number of operations of this feature implementation.

Likewise, Edicts implementation tangles driver and feature code considering aspects, as illustrated in Figure 4.11. At least one concrete subaspect has code tangling due to Edicts structure. The situation is even worse for feature implementation that uses more than one aspect, such as *Memory Budget*. On the other hand, the other three idioms implement a separated aspect that contains only the driver mechanism. Therefore, the *DOTC* is zero, except for the *Notation* feature, which presents similar implementations for all idioms, as we explained in Section 4.2.2.

Notice that *DOTO* and *DOTC* are zero for features seven *NextPiece* and eight *Record*

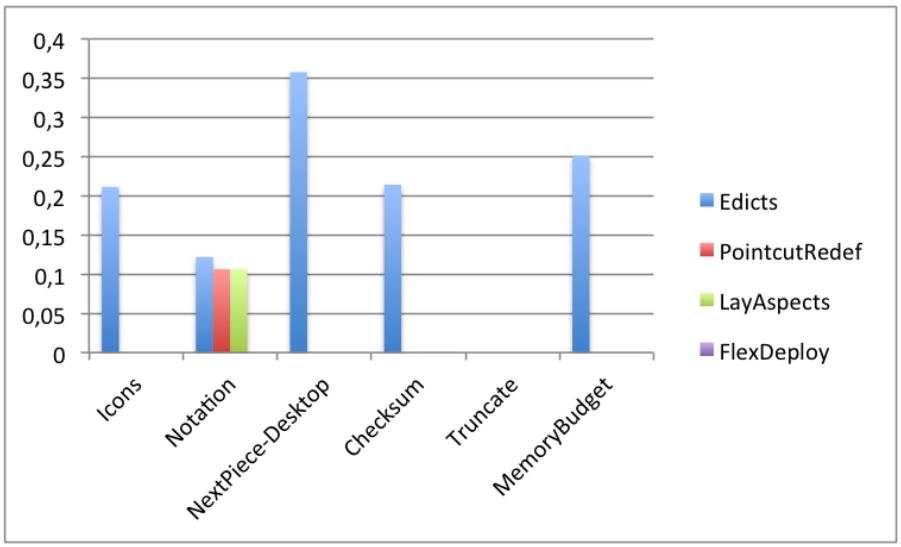



**Figure 4.11** DOTC metric results

because we implement them to mobile platform which only binds feature statically, so there is no driver. In addition, *DOTO* and *DOTC* are zero for the *Truncate* feature too because it does not implement pieces of advice, so there is no driver code too.

### 4.2.4 Size

To identify the idiom that reduces the size of its implementation, we try to answer the fourth question, which is related to the size of each idiom in terms of lines of code and the number of components. For this purpose, we use the *SLOC* and *VS* metrics.

As explained in Section 3.1, in order to support flexible binding time of a feature, the Edicts idiom introduces two additional concrete subaspects for each aspect implementing the feature. This situation leads to higher *SLOC* rates, mainly due to the code duplication introduced by dynamic and static aspects. Table 4.3 presents the *SLOC* metric results. In general, the other three idioms achieve better results than Edicts due to the absence of feature and driver code duplication. The difference becomes higher when the feature is large and contains many advice, which will be duplicated among the concrete subaspects. For example, *SLOC* presents high rates for Edicts to the *Memory Budget* feature comparing to the other idioms because it is large and contains many pieces of advice. In contrast, the *Truncate* feature does not contain pieces of advice and consequently, the Edicts idiom does not duplicate them, so the *SLOC* of Edicts is not the highest.

Now, we analyze the size from the number of components perspective by using the *VS* metric. We detail the results in Table 4.4. Unlike the other graphs, here we

**Table 4.3** SLOC metric results

| Feature | Edicts | PointcutRedef | Layered Aspects | FlexDeploy |
|---|---|---|---|---|
| **Icons** | 2198 | 2180 | 2127 | 2031 |
| **Clouds** | 2015 | 1919 | 1897 | 1833 |
| **Notation** | 172 | 172 | 172 | 153 |
| **Guillemets** | 320 | 208 | 181 | 178 |
| **NextPiece-Desktop** | 516 | 515 | 506 | - |
| **NextPiece-Mobile** | 450 | 449 | 435 | - |
| **Record-Desktop** | 511 | 507 | 387 | - |
| **Record-Mobile** | 460 | 335 | 455 | - |
| **EnvironmentLock** | 160 | 119 | 116 | 118 |
| **Checksum** | 476 | 469 | 441 | 456 |
| **Truncate** | 157 | 157 | 157 | 178 |
| **Delete** | 460 | 358 | 338 | 358 |
| **LookAheadCache** | 140 | 146 | 134 | 134 |
| **Evictor** | 452 | 466 | 460 | 470 |
| **NIO** | 61 | 52 | 52 | 52 |
| **MemoryBudget** | 1891 | 1397 | 1397 | 1285 |
| **IO** | 74 | 66 | 66 | 62 |
| **INCompressor** | 592 | 584 | 584 | 570 |

illustrate the difference between the case studies. Most of the case studies use a similar number of components to implement the features. However, the small differences happens because the Flexible Deployment implementation presents wrapper classes (Section 3.2.3), so it presents more components than the other idioms for Freemind, ArgoUML, and BerkeleyDB. The Tetris case study has few aspect on implement its features, then the *VS* results are equal to Edicts, Pointcut Redefinition, and Layered Aspects.

**Table 4.4** VS metric results

| Application | Edicts | PointcutRedef | Layered Aspects | FlexDeploy |
|---|---|---|---|---|
| **Freemind** | 550 | 551 | 549 | 557 |
| **ArgoUML** | 1622 | 1621 | 1621 | 1629 |
| **Tetris** | 21 | 21 | 21 | - |
| **Berkeley** | 345 | 338 | 345 | 350 |

### 4.2.5 Behavior

To obtain reliability that the execution of the implementation of flexible binding time does not change when implementing the same feature code with different idioms, we use the SafeRefactor [SGSM10] tool[3]. It automatically generates a test suite for two different

[3]http://code.google.com/p/saferefactoraj/downloads/detail?name=saferefactor.0.1.6.jar

versions (e.g., before and after the refactoring) of a given target program and reports whether it presents behavioral changes between them.

This way, we provide two different versions of the same application as inputs to SafeRefactor. These versions differ in the idiom used to implement flexible binding time for the features of this application. For example, we provide one version of the BerkeleyDB case study with Edicts and other version with Layered Aspects. Thus, SafeRefactor generates and execute a test suite to determine whether there is behavioral changes between the two versions. The Table 4.5 summarizes our results. Notice that SafeRefactor generates several unit tests for each of the four case studies. In addition, it generates the same number of tests for the source and target projects. For example, it generates 2005 tests for Freemind implemented with Edicts (source) and 2005 tests for Freemind implemented with Pointcut Redefinition (target). Then, it checks if the test results of both implementations are equal. If they are equal, then it informs that no behavioral changes are found.

We did not find behavioral changes in the Freemind and Tetris case studies. However, the SafeRefactor tool found some behavioral changes in the BerkeleyDB case study. Based on its results, we could identify that we had removed accidentally a line of code within a constructor defined in the `FileManager` class in the base code. Thereby, we could fix such inconsistence and increase the reliability of our implementations.

Unfortunately, SafeRefactor does not support the CaesarJ language. Hence, we could not test the Flexible Deployment implementation. However, we have not so far detected behavioral changes when using the applications implemented with this idiom.

**Table 4.5** Number of unit tests generated by SafeRefactor

| Application | Edicts and PcRedef | Edicts and LayAspects | PcRedef and LayAspects |
|---|---|---|---|
| **Freemind** | 2005 | 2072 | 2072 |
| **ArgoUML** | 1443 | 1443 | 1443 |
| **Tetris** | 2394 | 2380 | 2384 |
| **BerkeleyDB** | 3362 | 3141 | 3141 |

### 4.2.6 Threats to validity

In this section, we discuss some threats to the validity of our study.

**Case Studies.** Our work is based on only four case studies, which could be a limiting factor. In addition, these case studies were not widely used. However, we believe the selected case studies cover different architecture and code complexity. Tetris is a small

application implemented in few layers that runs for mobile and desktop environments. Freemind, ArgoUML, and BerkeleyDB are widely used applications. They are large systems implemented in many layers. Their functionalities are scattered throughout these layers.

**Selected Features.** Our selected features do not approach all existing architectures, granularity, and levels of complexity. Nevertheless, we modularized six feature for Freemind, ArgoUML, and Tetris, and we selected 12 features of the BerkeleyDB product line. They vary on the way they were implemented, on size, and on granularity. In addition, regarding feature model, we have mandatory, optional, OR, and alternative features, which turned possible to identify differences between them, and consequently improve our idioms in order to work with all these types of feature.

**Application refactoring.** To provide flexible binding times, we need to refactor the applications. As explained in Section 4.1.3, we first modularize the feature in cases which it is not modularized before, and then we implement the idiom. In this work, we only use the SafeRefactor [SGSM10] tool to validate refactoring. As explained in Section 4.2.5, this tool does not support the CaesarJ language. Thus, we only validate refactoring by performing user interface tests of the applications implemented with Flexible Deployment idiom.

**Multiple drivers.** In this work, we only consider applying one driver at a time. However, we realize that some applications may depend on several conditions to activate or deactivate a certain feature. For instance, Lee et al. utilize a home service robot product line as case study [LK06]. This robot changes its configuration dynamically depending on the environment brightness or its remaining battery. It would demand at least two drivers to (de)activate some of its features in our context. It may appear some problems, such as more driver tangling and scattering.

**Metrics.** Someone could argue that metrics do not cover all software code attributes affecting the system modularity. However, to improve the confidence in our evaluation, we gathered our conclusions based on a set of metrics, which considers eight metrics with respect to clone, modularity, and size. In addition, there is no previous work that considers evaluating the implementation of flexible binding time regarding code cloning, modularity, and size.

**Cloning results.** Our results for the *PCC* metric are based on the same parameters (token length and token set size) for all the four case studies, which could harm these results. Additionally, we do not use a manual filter, in which we would eliminate the cloned code that the tool detected but it is not interesting to our context. However, we

actually looked at the code to confirm that the *PCC* results show what idiom presents more cloned code. Indeed, it was apparent that Edicts presented more cloned code than the other idioms.

### 4.2.7 Discussion

In this section, we discuss the idioms qualitatively. The Edicts idioms may cause modularity problems in the most of the cases. It clones feature code because it duplicates pieces of advice among the subaspects of its structure. Further, it scatters and tangles driver code throughout these pieces of advice. In addition, its implementation size tends to be larger, since it clones part of the feature code. These problems are harmful to software maintenance. For instance, if we forget to introduce the driver mechanism into a piece of advice, we could have a run-time exception in case the feature is deactivated. Besides, maintaining the feature code may be time consuming and error prone due to the code cloning and scattering. For example, we have to fix the same problem twice for both concrete subaspects.

In contrast, Pointcut Redefinition does not clone feature code because it is modularized in the abstract aspect, therefore, it is not duplicated among the subaspects. However, it scatters driver code throughout the pointcuts redefined in the concrete subaspect. This could be harmful when adding, updating or removing the driver mechanism. If the feature implementation presents many pointcuts, we would have to change driver code in several pointcuts. In addition, Pointcut Redefinition does not tangle feature and driver code because the concrete subaspect that implements the driver mechanism does not contain feature code.

Moreover, Layered Aspects does not clone feature code too. Unlikely Pointcut Redefinition, this idiom only scatters driver code when `around` advice are present. This way, we mitigate the problem of changing the driver mechanism aforementioned. However, Layered Aspects does not solve it completely for such case because it still has to scatter driver code throughout the redefined pointcuts. In addition, this idiom does not tangle feature and driver code because neither of the concrete subaspects contain feature code. The feature code is implemented only in the abstract aspect, which does not contain driver code.

To sum up, Flexible Deployment does not clone, scatter or tangle feature and driver code. Nevertheless, CaesarJ does not support some AspectJ constructors we use to implement features. For example, we face problems to implement two features from Tetris because a class that is inherited from another may vary between desktop and

mobile platforms. Since CaesarJ does not modularize inheritance of classes, we could not implement these features without introducing a bias in the evaluation due to the several changes that we would have to do in Tetris. In addition, CaesarJ does not support privileged access for non-public members. Thus, we have to change the visibility of some methods in order to access them from within the CaesarJ virtual classes.

The implementations of flexible binding time using the four idioms were made by the same person to avoid introducing a bias. This way, the code of the features were modularized in the same way, and each idiom was applied following its design for the 18 features of the four applications.

Nevertheless, our idioms may present some deficiencies. For example, the (de)activation of one feature may need the (de)activation of other features. In the dynamic binding time context, if we do not validate feature composition when applying changes, it may lead to run-time errors and possibly compromise the program execution. In this work, we do not validate the composition of features dynamically. Thereby, our idioms may present deficiencies when (de)activating features at run-time. This could happen in the BerkeleyDB case study. For example, the activation of the *Truncate* feature implies the activation of the *Delete* feature. Hence, if we use one of our idioms to dynamically deactivate *Delete*, then the *Truncate* feature will not work properly. We plan to circumvent this issue in future work.

In addition, previous works documented a more efficient execution of systems for static binding rather than dynamic binding [RSSA08, GS10]. Our work does not include an evaluation regarding performance and memory consumption. Thus, we do not know whether our idioms improve or deteriorate the feature code execution. Nevertheless, we do not use techniques or constructs that might harm performance or memory consumption [CRE08]. We do not notice significant differences when using the applications.

Besides that, we only use aspect-oriented based idioms to implement flexible binding time for features in applications written in Java. This way, some disadvantages of our idioms may appear due to these technologies. For example, in Section 3.2.2, we explain that AspectJ does not provide access to the `proceed` join point of the advice. This is a disadvantage since we have to deactivate the pieces of advice one by one instead of all of them once.

Finally, the implementations of Edicts, Layered Aspects, and Pointcut Redefinition do not present behavioral changes according to SafeRefactor. We aim to avoid changing the behavior of the applications so that the only modification is the implementation of flexible binding time. Unfortunately, SafeRefactor does not support CaesarJ, then we

could not run it for the Flexible Deployment idiom. However, we run the applications to verify changes in their behavior and we do not find any.

# 5 Conclusion

In this work, we discuss the implementation of flexible binding time for features. With this purpose, we introduce three idioms to address the shortcomings presented by an idiom called Edicts. Our idioms mitigate some issues, such as code cloning, scattering, and tangling.

In this context, the Pointcut Redefinition idiom uses inheritance of AspectJ aspects and redefinition of pointcuts to implement flexible binding time. We modularize feature code in an abstract aspect and implement the static and dynamic binding time in different concrete subaspects. The subaspect that implements static binding is empty and inherits the abstract aspect to allow its instantiation. The subaspect that implements dynamic binding redefines the pointcuts defined in the abstract aspect and associate them to the driver mechanism. This idiom mitigates the code cloning and tangling presented by Edicts. However, it does not reduce driver code scattering significantly because we have to redefine the pointcuts one by one and associate them with the driver mechanism.

The second idiom, Layered Aspects, is based on AspectJ. We modularize feature code in an abstract aspect and implement static binding in the same way we do for Pointcut Redefinition. To implement dynamic binding time, we define a concrete subaspect that uses the AspectJ `adviceexecution` in association with redefinition of pointcuts. Layered Aspects reduce code cloning, scattering, tangling, and size comparing to Edicts.

The third idiom, Flexible Deployment, uses the dynamic deployment mechanism provided by CaesarJ to implement binding time flexibility. We modularize feature code in a CaesarJ class and implement the static and dynamic binding time in different CaesarJ classes. To implement static binding, we define an empty CaesarJ subclass to allow the instantiation of the class that contains feature code. To implement dynamic binding, we define a CaesarJ class that contains one pointcut and one advice, they implement the dynamic deployment mechanism. Despite of we could not apply Flexible Deployment to

four of our 18 features, this idiom presents good results regarding the metrics. It reduces code cloning, scattering, and tangling comparing to Edicts.

Additionally, this work evaluates these idioms by means of metrics. To achieve representative results, we use the four idioms to implement flexible binding time for 18 features of four different case studies. Our evaluation is based on the GQM approach. Thus, we define goals, questions, and metrics to assess the idioms for each one of the 18 features. Furthermore, we illustrate an example of interaction between features, such as a group of alternative features and how to apply our idioms for such situations. We also discuss the idioms qualitatively regarding advantages and disadvantages of each one.

Finally, this work presents some known threats we wish to address as future work such as safe dynamic composition of features and implementation of idioms with multiple drivers.

## 5.1 Related work

Besides Edicts, we point out other work regarding flexible binding times as well as studies that relate aspects and product line features. Additionally, we discuss how our work differ from them.

**Tailoring Dynamic Software Product Lines.** Rosenmuller et al. propose an approach of statically generating tailor-made dynamic software product lines (DSPL) from SPL features [RSPA11]. Besides, the authors describe a feature-based approach of adaptation and self-configuration to ensure composition safety. They statically select the features required for dynamic binding and generate a set of binding units that are composed at run-time to yield the program. Additionally, they implement their approach in one case study and evaluate it with concern to reconfiguration performance at run-time.

In contrast, we believe that our evaluation (Section 4.2) covers more important aspects. It is relevant to actually evaluate the approaches with respect to modularity, size, and code cloning. This way, we avoid undesired solutions that are hard to maintain, for example. We also provide all the source code and metric sheets to permit researchers to replicate our work.

**Code Generation to Support Static and Dynamic Composition of Software Product Lines.** Rosenmuller et al. present an approach that supports static and dynamic composition of features from a single code base [RSSA08]. They provide an infrastructure for dynamic instantiation and validation of SPLs. Their approach is based on FOP [Pre97], and it uses an FOP extension called FeatureC++ [ALRS05] to automate

dynamic instantiation of SPLs.

The usage of C++ as a client language introduces some specific problems. Static constructs when using dynamic composition, virtual classes, semantic differences when comparing static and dynamic compositions are examples of such problems [RSSA08]. Despite of our work uses only Java as a client language, we did not observe these problems in our implementations. Furthermore, the authors only evaluate the approach regarding to performance, applicability and memory consumption. In our work, we have a wider evaluation as showed in Section 4.2.

**A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering.** Lee et al. propose a systematic approach to develop dynamically reconfigurable core assets [LK06]. Besides, they present strategies to manage product configuration at run-time.

This work only consider dynamic binding time whereas we take into account dynamic and static binding time. Additionally, the authors utilize a home service robot product line as an example of their dynamically reconfigurable product. Nevertheless, differently of us, they do not evaluate their approach considering software modularity or code quality.

**The Variability of Binding Time of Variation Points.** An alternative proposal considers *conditional compilation* as a technique to implement features with flexible binding times [UFD03]. This work discusses how to apply *conditional compilation* in real applications like operating systems. Likewise we describe in our work, developers need to decide what features should be included to compose the product and their respective binding times.

However, the work concludes that, in fact, conditional compilation is not a very elegant solution to flexible the binding time. Hence, for complex variation points, the situation becomes even worse.

**Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines.** Trinidad et al. propose a process to generate a component architecture that is able to activate or deactivate features dynamically and performs some analysis operations on feature models to ensure that the feature composition is valid [TCPB07]. They apply their approach to generate an industrial real-time television SPL.

This work provides only dynamic binding time. It aim at adapting systems to changing requirements. This way, they do not consider the performance overhead that dynamic binding should introduce [RSSA08]. Besides, there is no evaluation of the approach.

**Aspect Inheritance.** Another proposal to implement flexible binding time into features considers aspect inheritance [RCB+09]. It defines an idiom that relies on aspect

inheritance through the abstract pointcut definition. This solution states that we have to create an abstract aspect with feature code and an abstract pointcut definition, then we associate this driver with the advice. Furthermore, we create two concrete subaspects inheriting from the abstract one in order to implement the concrete driver. Differently from Edicts, Aspect Inheritance avoids feature code duplication. However, this solution is not suitable, because despite of avoiding feature code cloning, the other results are similar to Edicts.

**Dynamic Software Product Line Approach using Aspect Models at Runtime.** Dinkelaker et al. [DMFM10] propose an approach that uses a dynamic feature model to describe variability and uses a domain-specific language for declaratively implementing variations and their constraints. Their approach has mechanisms to detect and resolve feature interactions dynamically by validating an aspect-oriented model at run-time.

**Composing Aspect with Aspects.** Marot et al. [MW10] propose OARTA, which is a declarative extension to the AspectBench Compiler [ACH+05], which allows dynamic weaving of aspects. OARTA extends the AspectJ language syntax so that a developer can name a pointcut, which allows referring to it later on. It is possible that aspects weave on other aspects. Therefore, they exemplify how to dynamically deactivate features in run-time situations (e.g. features competing for resources, which may be deactivated to speed up the execution). By using AspectJ, we would have to add an if(..) pointcut predicate to the pointcut of the advice that contains feature code. However, they do not perform an assessment considering quality of code factors or modularity of their approach. Moreover, they only consider dynamic binding time of features.

## 5.2 Future work

As explained in Section 4.2.6, we do not consider dynamic safe composition of features. Our applications may present some problems we (de)activating some features. For example, the features NIO and IO from BerkeleyDB are alternatives. Thus, if NIO is activated and we activate IO, BerkeleyDB will not work correctly because only one of the features can be activated at a given time. Hence, we intend to support dynamic safe composition of features. We consider two alternatives for solving this issue. The first one is trying to integrate the domain-specific language proposed by Dinkelaker et al. introduced in Section 5.1. The second one is creating a new technique to contemplate our idioms. We still need to evaluate both alternatives and decide which one is better for our context.

Another possible future work is to evaluate the performance and memory consumption comparing static and dynamic binding time, and considering different idioms. To do this, we may analyze the consumption of working memory by observing the size of instances of the aspects that implement feature and binding time. Furthermore, we intend to compare static and dynamic binding time, and the distinct idioms with respect to performance. The results could lead in new conclusions about what idiom we may adopt. This way, we would have a more complete evaluation, not only considering code quality, but also regarding to the code execution.

Besides, we intend to include applications that use multiple driver scenarios. This would increase the representativeness of our study because it may reveal problems, such as driver code scattering and tangling. Additionally, we may investigate addition and removal of drivers. We believe it may lead to complex situations. For example, we apply the driver mechanism in Edicts by introducing `if` statements in pieces of advice. Thus, it should be problematic to add new driver conditions because we would increase the complexity of the expressions evaluated in these `if` statements. Furthermore, it turns the driver code maintenance error-prone and time consuming.

It is also our intent to define an idiom not based on AOP. We could mitigate some problems related to this paradigm, such as the one presented in Section 3.2.2. Additionally, our case studies are written in Java, we do not consider other language in this work. Thereby, we also could implement a new idiom based on feature-oriented programming and the C++ language, similar to what Rosenmuller et al. do in their work [RSPA11]. Since there is no assessment in related work that evaluates the results regarding to metrics, we may achieve some interesting results.

Last but not least, we may add other metrics to our metric suite. Chidamber et al. define the Coupling Between Components [CK94] metric that we could use to count the number of classes or aspects declaring methods, constructors or fields that are possibly called or accessed by another aspect or class. This metric could show opportunities to mitigate the coupling of feature modularization, for example. In addition, we intend to use the Lack of Cohesion over Operations metric [CT04]. Thus, it could show what idiom has low code cohesion and consequently identify difficult reuse, maintenance, and understanding of code.

# Bibliography


[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *Proceedings of the 4th international conference on Aspect-oriented software development*, AOSD '05, pages 87–98, New York, NY, USA, 2005. ACM.

[AEKHG05] Raihan Al-Ekram, Cory Kapser, Richard Holt, and Michael Godfrey. Cloning by accident: An empirical study of source code cloning across software systems. In *Across Software Systems.International Symposium on Empirical Software Engineering (ISESEÕ05*, pages 376–385, 2005.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, pages 135–173, 2006.

[ALRS05] Sven Apel, Thomas Leich, Marko Rosenmuller, and Gunter Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *In Proceedings of the International Conference on Generative Programming and Component Engineering*, pages 125–140. Springer, 2005.

[Ape07] S. Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, School of Computer Science, University of Magdeburg, March 2007.

[Arg09] ArgoUML. Argouml, October 2009. http://argouml.tigris.org/.

[ARG+11] Rodrigo Andrade, Marcio Ribeiro, Vaidas Gasiunas, Lucas Satabin, Henrique Rebelo, and Paulo Borba. Assessing idioms for implementing features with flexible binding times. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, pages 231–240, Oldenburg, DE, 2011. IEEE Computer Society.

[BB09] Rodrigo Bonifácio and Paulo Borba. Modeling scenario variability as crosscutting mechanisms. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD'09)*, pages 125–136. ACM, 2009.

[BBM96] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.

[BCR94] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach. The goal question metric approach. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 528–532. Wiley, 1994.

[Ber10] Berkeley. Oracle berkeley db, April 2010. http://www.oracle.com/technology/products/berkeley-db/index.html.

[BvDvET05] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31:804–818, October 2005.

[BYM+98] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 368–377, 1998.

[ccf07] CCfinder Official Site, May 2007. http://www.ccfinder.net/.

[CHOT99] Siobhán Clarke, William H. Harrison, Harold Ossher, and Peri L. Tarr. Separating concerns throughout the development lifecycle. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 299–, London, UK, 1999. Springer-Verlag.

[CK94] S.R. Chidamber and C.F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[CRE08] Venkat Chakravarthy, John Regehr, and Eric Eide. Edicts: Implementing Features with Flexible Binding Times. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 108–119, New York, NY, USA, 2008. ACM.

[CT04] M. Ceccato and P. Tonella. Measuring the effects of software aspectization. In *1st Workshop on Aspect Reverse Engineering*, 2004.

[DHJ+08] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. Clone detection in automotive model-based development. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 603–612, New York, NY, USA, 2008. ACM.

[DMFM10] Tom Dinkelaker, Ralf Mitschke, Karin Fetzer, and Mira Mezini. A dynamic software product line approach using aspect models at runtime. In *Proceedings of the 1st Workshop on Composition and Variability*, March 2010.

[Ead08] Marc Eaddy. *An Empirical Assessment of the Crosscutting Concern Problem*. PhD thesis, Graduate School of Arts and Sciences, Columbia, USA, 2008.

[EAM07] Marc Eaddy, Alfred Aho, and Gail C. Murphy. Identifying, assigning, and quantifying crosscutting concerns. In *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, ACoM '07, pages 2–, Washington, DC, USA, 2007. IEEE Computer Society.

[EZS+08] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[FGS+05] E. Figueiredo, A. Garcia, C. Sant'anna, U. Kulesza, and C. Lucena. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. In *Proc. of the 9th ECOOP Workshop on Quantitative Approaches in OO Soft. Engineering (QAOOSE. 05)*, Glasgow, July 2005.

[Fre09] Freemind. Free mind mapping software, July 2009. http://freemind.sourceforge.net/.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, January 1995.

[GS10] Sebastian Günther and Sagar Sunkle. Dynamically adaptable software product lines using ruby metaprogramming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 80–87, New York, NY, USA, 2010. ACM.

[GSF+05] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, New York, NY, USA, March 2005. ACM Press.

[GSF+06] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *LNCS Transactions on Aspect-Oriented Software Development I*, pages 36–74. Springer, 2006.

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[KAB07] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*, pages 223–232, Washington, DC, USA, 2007. IEEE Computer Society.

[KCH+90] Kyo-Chul Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA). Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.

[KG06] Cory J. Kapser and Michael W. Godfrey. Supporting the analysis of clones in software systems: A case study. *JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE*, 18:2006, 2006.

[KHH+01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[KHH+01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[KKI02] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. CCFinder: A Multi-Linguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Engineering*, 28(7):654–670, 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect–Oriented Programming. In *Proceedings of European Conference on Object–Oriented Programming (ECOOP'97)*, LNCS 1241, pages 220–242, 1997.

[Kru92] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24:131–183, June 1992.

[KSG+06] Uira Kulesza, Claudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proceedings of the 22th IEEE International Conference on Software Maintenance (ICSM'06)*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society.

[LB05] Cristina Videira Lopes and Sushil Krishna Bajracharya. An Analysis of Modularity in Aspect-Oriented Design. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 15–26, New York, NY, USA, March 2005. ACM Press.

[LK06] Jaejoon Lee and Kyo C. Kang. A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In *Proceedings of the 10th International on Software Product Line Conference*, pages 131–140, Washington, DC, USA, 2006. IEEE Computer Society.

[LLO03] Karl Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46:2003, 2003.

[MLWR01] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: an exploratory study. In *Proceedings*

*of the 23rd International Conference on Software Engineering*, ICSE '01, pages 275–284, Washington, DC, USA, 2001. IEEE Computer Society.

[MW10] Antoine Marot and Roel Wuyts. Composing aspects with aspects. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD '10, pages 157–168, New York, NY, USA, 2010. ACM.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.

[PBvdL05] Klaus Pohl, Gunter Bockle, and Frank J. van der Linden. *Software Product Line Engineering*. Springer, 2005.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, pages 419–443, 1997.

[RCB+09] Márcio Ribeiro, Rodrigo Cardoso, Paulo Borba, Rodrigo Bonifácio, and Henrique Rebêlo. Does aspectj provide modularity when implementing features with flexible binding times? In *Third Latin American Workshop on Aspect-Oriented Software Development (LA-WASP 2009)*, pages 1–6, Fortaleza, Cear'a, Brazil, 2009.

[RSPA11] Marko Rosenmüller, Norbert Siegmund, Mario Pukall, and Sven Apel. Tailoring dynamic software product lines. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE '11, pages 3–12, New York, NY, USA, 2011. ACM.

[RSSA08] Marko Rosenmüller, Norbert Siegmund, Gunter Saake, and Sven Apel. Code generation to support static and dynamic composition of software product lines. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*, pages 3–12, New York, NY, USA, 2008. ACM.

[SGSM10] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52 –57, july-aug. 2010.

[SLB02] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM*

*SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'2002*, pages 174–190, Seattle, USA, 4th–8th November 2002.

[SR05] Damith Rajapakse School and Damith C. Rajapakse. An investigation of cloning in web applications. In *In Proceedings of the Special Interest Tracks and Posters of the 14th International Conference on World Wide Web (WWWÕ05*, pages 924–925. Springer, 2005.

[TBD06] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 191–200, New York, NY, USA, 2006. ACM.

[TCPB07] Pablo Trinidad, Antonio Ruiz Cortés, Joaquín Peña, and David Benavides. Mapping feature models onto component models to build dynamic software product lines. In *SPLC (2)*, pages 51–56, 2007.

[Tet09] Tetris. Tetris, July 2009. http://kiang.org/jordan/software/tetrismidlet/.

[UFD03] Eelco Utrecht, Gert Florijn, and Eelco Dolstra. Timeline variability: The variability of binding time of variation points. In *Proceedings of the Workshop on Software Variability Management (SVM'03)*, pages 119–122, 2003.

[vO02] R. van Ommering. Building product populations with software components. *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 255 – 265, 2002.

# A

# Appendix A - Implementation of idioms

In this appendix, we present the implementation of flexible binding time for the BerkeleyDB feature *Checksum*. This feature implementation detects corruptions when writing or reading a database page. Each one of the following sections, correspond to a different idiom.

## A.1 Edicts

**Listing A.1** ChecksumDynamic.

```
1  package com.checksum;
2
3  import java.io.RandomAccessFile;
4  import java.nio.ByteBuffer;
5  import com.sleepycat.je.DatabaseException;
6  import com.sleepycat.je.cleaner.FileProcessor;
7  import com.sleepycat.je.dbi.EnvironmentImpl;
8  import com.sleepycat.je.log.EntryHeader;
9  import com.sleepycat.je.log.FileManager;
10 import com.sleepycat.je.log.FileReader;
11 import com.sleepycat.je.log.LogManager;
12 import com.sleepycat.je.log.PrintFileReader;
13 import com.sleepycat.je.log.entry.LogEntry;
14 import com.sleepycat.je.recovery.RecoveryManager;
15 import com.sleepycat.je.log.*;
16 import com.sleepycat.je.log.LogSource;
17
18 public privileged abstract aspect ChecksumAbstract {
19
20  pointcut fileReaderConstructor(FileReader fileReader, EnvironmentImpl env) : execution
        (FileReader.new(EnvironmentImpl, int, boolean, long, Long, long, long)) && args(
        env,int, boolean, long, Long, long,long) && this(fileReader) && within(FileReader)
        ;
21
```

```
22 pointcut hook_checksumValidation(FileReader fileReader, ByteBuffer dataBuffer) : call(
       void FileReader.hook_checksumValidation(ByteBuffer)) && target(fileReader) && args
       (dataBuffer);
23
24 pointcut hook_checkType(FileReader fr, byte currentEntryTypeNum) : execution(void
       FileReader.hook_checkType(byte)) && args(currentEntryTypeNum) && this(fr);
25
26 pointcut logManagerConstructor(LogManager logManager, EnvironmentImpl env) : execution
       (LogManager.new(EnvironmentImpl, boolean)) && this(logManager) && args(env,boolean
       );
27
28 pointcut addPrevOffset(int entrySize) : execution(ByteBuffer LogManager.addPrevOffset(
       ByteBuffer,long,int)) && args(ByteBuffer,long,entrySize) && within(LogManager);
29
30 pointcut readHeader(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader) :
       call(void EntryHeader.readHeader(ByteBuffer, boolean)) && this(lm) && args(
       entryBuffer,boolean) && target(entryHeader);
31
32 pointcut getRemainingBuffer(LogManager lm, ByteBuffer entryBuffer, EntryHeader
       entryHeader, long lsn) : call(ByteBuffer LogManager.getRemainingBuffer(ByteBuffer,
        LogSource, long, EntryHeader)) && target(lm) && args(entryBuffer, LogSource, long
       , entryHeader) && cflow(getLogEntryFromLogSource(lsn));
33
34 pointcut readHeader2(EntryHeader eh, ByteBuffer dataBuffer) : execution(void
       EntryHeader.readHeader(ByteBuffer,boolean)) && target(eh) && args(dataBuffer,
       boolean);
35
36 pointcut lasFileReaderConstructor(FileReader fr) : execution(LastFileReader.new(..))
      && this(fr);
37
38 pointcut iNFileReaderConstructor() : (call(INFileReader.new(..)) && withincode(void
       RecoveryManager.readINsAndTrackIds(long))) || (call(CleanerFileReader.new(..)) &&
       within(FileProcessor));
39
40 pointcut hook_recomputeChecksum(ByteBuffer data, int recStartPos, int itemSize) : call
       (void FileManager.hook_recomputeChecksum(ByteBuffer, int, int)) && args(data,
       recStartPos,itemSize);
41
42 pointcut readNextEntry() : execution(boolean LastFileReader.readNextEntry());
43
44 pointcut readAndValidateFileHeader(RandomAccessFile newFile, String fileName,
       FileManager fm) : execution(boolean FileManager.readAndValidateFileHeader(
       RandomAccessFile, String, long)) && args(newFile,fileName,long) && target(fm) &&
       within(FileManager);
45
46 pointcut readHeader3(ByteBuffer dataBuffer, FileReader fileReader) : execution(void
       FileReader.readHeader(ByteBuffer)) && args(dataBuffer) && target(fileReader);
47
48 pointcut allocate() : call(ByteBuffer ByteBuffer.allocate(int)) && withincode(
       ByteBuffer LogManager.marshallIntoBuffer(LoggableObject, int, boolean, int));
49
50 pointcut hook_printChecksum(StringBuffer sb, PrintFileReader pfr) : call(void
       PrintFileReader.hook_printChecksum(StringBuffer)) && args(sb) && this(pfr);
51
```

```
52  private pointcut getLogEntryFromLogSource(long lsn) : execution(LogEntry LogManager.
        getLogEntryFromLogSource(long,LogSource)) && args(lsn, LogSource);
53
54  public ChecksumValidator FileReader.cksumValidator;
55
56  private boolean FileReader.doValidateChecksum;
57
58  private boolean FileReader.alwaysValidateChecksum;
59
60  public boolean FileReader.anticipateChecksumErrors;
61
62  private void FileReader.startChecksum(ByteBuffer dataBuffer) throws DatabaseException
        {
63   cksumValidator.reset();
64   int entryStart = threadSafeBufferPosition(dataBuffer);
65   dataBuffer.reset();
66   cksumValidator.update(env, dataBuffer, LogManager.HEADER_CONTENT_BYTES,
         anticipateChecksumErrors);
67   threadSafeBufferPosition(dataBuffer, entryStart);
68  }
69
70  public long FileReader.currentEntryChecksum;
71
72  private void FileReader.validateChecksum(ByteBuffer entryBuffer) throws
        DatabaseException {
73
74   cksumValidator.update(env, entryBuffer, currentEntrySize, anticipateChecksumErrors);
75   cksumValidator.validate(env, currentEntryChecksum, readBufferFileNum,
         currentEntryOffset, anticipateChecksumErrors);
76  }
77
78  public void FileReader.setAlwaysValidateChecksum(boolean validate) {
79   alwaysValidateChecksum = validate;
80  }
81
82  private boolean LogManager.doChecksumOnRead;
83
84  public boolean LogManager.getChecksumOnRead() {
85   return doChecksumOnRead;
86  }
87
88  private ChecksumValidator validator = null;
89
90  private long EntryHeader.checksum;
91
92  public long EntryHeader.getChecksum() {
93   return checksum;
94  }
95
96  static final int LogManager.CHECKSUM_BYTES = 4;
97
98  static final int LogManager.HEADER_CHECKSUM_OFFSET = 0;
99 }
```

**Listing A.2** ChecksumStatic.

```
1  package com.checksum;
2
3  import java.io.RandomAccessFile;
4  import java.nio.ByteBuffer;
5  import java.util.zip.Checksum;
6
7  import com.sleepycat.je.DatabaseException;
8  import com.sleepycat.je.config.EnvironmentParams;
9  import com.sleepycat.je.dbi.DbConfigManager;
10 import com.sleepycat.je.dbi.EnvironmentImpl;
11 import com.sleepycat.je.log.EntryHeader;
12 import com.sleepycat.je.log.FileManager;
13 import com.sleepycat.je.log.FileReader;
14 import com.sleepycat.je.log.LogEntryType;
15 import com.sleepycat.je.log.LogManager;
16 import com.sleepycat.je.log.LogUtils;
17 import com.sleepycat.je.log.PrintFileReader;
18
19 public privileged aspect ChecksumValidatorAspectStatic extends ChecksumAbstract {
20
21  Object around(FileReader fileReader, EnvironmentImpl env) throws DatabaseException :
        fileReaderConstructor(fileReader, env) {
22   fileReader.doValidateChecksum = env.getLogManager().getChecksumOnRead();
23   Object r = proceed(fileReader, env);
24   if (fileReader.doValidateChecksum) {
25    fileReader.cksumValidator = new ChecksumValidator();
26   }
27   fileReader.anticipateChecksumErrors = false;
28   return r;
29  }
30
31  after(FileReader fileReader, ByteBuffer dataBuffer) throws DatabaseException :
        hook_checksumValidation(fileReader, dataBuffer){
32   boolean doValidate = fileReader.doValidateChecksum && (fileReader.
         alwaysValidateChecksum || fileReader .isTargetEntry(fileReader.
         currentEntryTypeNum, fileReader.currentEntryTypeVersion));
33   if (doValidate) {
34    fileReader.startChecksum(dataBuffer);
35   }
36   if (doValidate)
37    fileReader.currentEntryCollectData = true;
38  }
39
40  before(FileReader fr, byte currentEntryTypeNum) throws DatabaseException :
        hook_checkType(fr, currentEntryTypeNum)      {
41   if (!LogEntryType.isValidType(currentEntryTypeNum))
42    throw new DbChecksumException((fr.anticipateChecksumErrors ? null : fr.env), "
          FileReader␣read␣invalid␣log␣entry␣type:␣" + currentEntryTypeNum);
43  }
44
45  after(LogManager logManager, EnvironmentImpl env) throws DatabaseException :
        logManagerConstructor(logManager, env) {
```

```
46   DbConfigManager configManager = env.getConfigManager();
47   logManager.doChecksumOnRead = configManager.getBoolean(EnvironmentParams.
         LOG_CHECKSUM_READ);
48  }
49
50  ByteBuffer around(int entrySize) : addPrevOffset(entrySize) {
51   ByteBuffer destBuffer = proceed(entrySize);
52   Checksum checksum = Adler32.makeChecksum();
53   checksum.update(destBuffer.array(), LogManager.CHECKSUM_BYTES, (entrySize -
         LogManager.CHECKSUM_BYTES));
54   LogUtils.writeUnsignedInt(destBuffer, checksum.getValue());
55   destBuffer.position(0);
56   return destBuffer;
57  }
58
59  after(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader) throws
        DatabaseException : readHeader(lm, entryBuffer, entryHeader) {
60   if (lm.doChecksumOnRead) {
61    validator = new ChecksumValidator();
62    int oldpos = entryBuffer.position();
63    entryBuffer.position(oldpos - LogManager.HEADER_CONTENT_BYTES);
64    validator.update(lm.envImpl, entryBuffer, LogManager.HEADER_CONTENT_BYTES, false);
65    entryBuffer.position(oldpos);
66   }
67  }
68
69  after(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader, long lsn) throws
         DatabaseException : getRemainingBuffer(lm, entryBuffer, entryHeader, lsn) {
70   if (lm.doChecksumOnRead) {
71    validator.update(lm.envImpl, entryBuffer, entryHeader.getEntrySize(), false);
72    validator.validate(lm.envImpl, entryHeader.getChecksum(), lsn);
73   }
74  }
75
76  before(EntryHeader eh, ByteBuffer dataBuffer): readHeader2(eh, dataBuffer) {
77   eh.checksum = LogUtils.getUnsignedInt(dataBuffer);
78  }
79
80  after(FileReader fr) : lasFileReaderConstructor(fr) {
81   fr.anticipateChecksumErrors = true;
82  }
83
84  after() returning (FileReader fr) : iNFileReaderConstructor() {
85   fr.setAlwaysValidateChecksum(true);
86  }
87
88  after(ByteBuffer data, int recStartPos, int itemSize) : hook_recomputeChecksum(data,
        recStartPos,itemSize) {
89   Checksum checksum = Adler32.makeChecksum();
90   data.position(recStartPos);
91   int nChecksumBytes = itemSize + (LogManager.HEADER_BYTES - LogManager.CHECKSUM_BYTES)
         ;
92   byte[] checksumBytes = new byte[nChecksumBytes];
```

```
93    System.arraycopy(data.array(), recStartPos + LogManager.CHECKSUM_BYTES, checksumBytes
          , 0, nChecksumBytes);
94    checksum.update(checksumBytes, 0, nChecksumBytes);
95    LogUtils.writeUnsignedInt(data, checksum.getValue());
96   }
97
98   boolean around() : readNextEntry() {
99    boolean r = false;
100   try {
101    r = proceed();
102   } catch (DbChecksumException e) {
103   }
104   return r;
105  }
106
107  boolean around(RandomAccessFile newFile, String fileName, FileManager fm) throws
         DatabaseException: readAndValidateFileHeader(newFile, fileName, fm) {
108   try {
109    return proceed(newFile, fileName, fm);
110   } catch (DbChecksumException e) {
111    fm.closeFileInErrorCase(newFile);
112    throw new DbChecksumException(fm.envImpl, "Couldn't␣open␣file␣" + fileName, e);
113   }
114  }
115
116  before(ByteBuffer dataBuffer, FileReader fileReader) : readHeader3(dataBuffer,
         fileReader) {
117   fileReader.currentEntryChecksum = LogUtils.getUnsignedInt(dataBuffer);
118  }
119
120  after(): staticinitialization(LogManager) {
121   LogManager.HEADER_BYTES += LogManager.CHECKSUM_BYTES;
122   LogManager.PREV_BYTES = 4;
123   LogManager.HEADER_CONTENT_BYTES = LogManager.HEADER_BYTES - LogManager.CHECKSUM_BYTES
          ;
124   LogManager.HEADER_ENTRY_TYPE_OFFSET += 4;
125   LogManager.HEADER_VERSION_OFFSET += 4;
126   LogManager.HEADER_PREV_OFFSET += 4;
127   LogManager.HEADER_SIZE_OFFSET += 4;
128  }
129
130  after()returning(ByteBuffer buffer) : allocate() {
131   buffer.position(LogManager.CHECKSUM_BYTES);
132  }
133
134  after(StringBuffer sb, PrintFileReader pfr) : hook_printChecksum(sb, pfr){
135   sb.append("\"␣cksum=\"").append(pfr.currentEntryChecksum);
136  }
137 }
```

**Listing A.3** ChecksumDynamic.

```
1 package com.checksum;
2
```

```
3  import java.io.RandomAccessFile;
4  import java.nio.ByteBuffer;
5  import java.util.zip.Checksum;
6
7  import com.sleepycat.je.DatabaseException;
8  import com.sleepycat.je.config.EnvironmentParams;
9  import com.sleepycat.je.dbi.DbConfigManager;
10 import com.sleepycat.je.dbi.EnvironmentImpl;
11 import com.sleepycat.je.log.EntryHeader;
12 import com.sleepycat.je.log.FileManager;
13 import com.sleepycat.je.log.FileReader;
14 import com.sleepycat.je.log.LogEntryType;
15 import com.sleepycat.je.log.LogManager;
16 import com.sleepycat.je.log.LogUtils;
17 import com.sleepycat.je.log.PrintFileReader;
18
19 import driver.Driver;
20
21 public privileged aspect ChecksumValidatorAspectStatic extends ChecksumAbstract {
22
23  Driver driver = new Driver();
24
25  Object around(FileReader fileReader, EnvironmentImpl env) throws DatabaseException :
        fileReaderConstructor(fileReader, env) {
26   if (driver.isActivated("checksum")) {
27    fileReader.doValidateChecksum = env.getLogManager().getChecksumOnRead();
28    Object r = proceed(fileReader, env);
29    if (fileReader.doValidateChecksum) {
30     fileReader.cksumValidator = new ChecksumValidator();
31    }
32    fileReader.anticipateChecksumErrors = false;
33    return r;
34   }
35   return proceed(fileReader, env);
36  }
37
38  after(FileReader fileReader, ByteBuffer dataBuffer) throws DatabaseException :
        hook_checksumValidation(fileReader, dataBuffer){
39   if (driver.isActivated("checksum")) {
40    boolean doValidate = fileReader.doValidateChecksum && (fileReader.
          alwaysValidateChecksum || fileReader .isTargetEntry(fileReader.
          currentEntryTypeNum, fileReader.currentEntryTypeVersion));
41    if (doValidate) {
42     fileReader.startChecksum(dataBuffer);
43    }
44    if (doValidate)
45     fileReader.currentEntryCollectData = true;
46    }
47  }
48
49  before(FileReader fr, byte currentEntryTypeNum) throws DatabaseException :
        hook_checkType(fr, currentEntryTypeNum) {
50   if (driver.isActivated("checksum")) {
51    if (!LogEntryType.isValidType(currentEntryTypeNum))
```

```
52      throw new DbChecksumException((fr.anticipateChecksumErrors ? null : fr.env), "
            FileReader read invalid log entry type: " + currentEntryTypeNum);
53    }
54   }
55
56   after(LogManager logManager, EnvironmentImpl env) throws DatabaseException :
         logManagerConstructor(logManager, env) {
57    if (driver.isActivated("checksum")) {
58     DbConfigManager configManager = env.getConfigManager();
59     logManager.doChecksumOnRead = configManager.getBoolean(EnvironmentParams.
           LOG_CHECKSUM_READ);
60    }
61   }
62
63   ByteBuffer around(int entrySize) : addPrevOffset(entrySize) {
64    ByteBuffer destBuffer = proceed(entrySize);
65    if (driver.isActivated("checksum")) {
66     ByteBuffer destBuffer = proceed(entrySize);
67     Checksum checksum = Adler32.makeChecksum();
68     checksum.update(destBuffer.array(), LogManager.CHECKSUM_BYTES, (entrySize -
           LogManager.CHECKSUM_BYTES));
69     LogUtils.writeUnsignedInt(destBuffer, checksum.getValue());
70     destBuffer.position(0);
71    }
72    return destBuffer;
73   }
74
75   after(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader) throws
         DatabaseException : readHeader(lm, entryBuffer, entryHeader) {
76    if (driver.isActivated("checksum")) {
77     if (lm.doChecksumOnRead) {
78      validator = new ChecksumValidator();
79      int oldpos = entryBuffer.position();
80      entryBuffer.position(oldpos - LogManager.HEADER_CONTENT_BYTES);
81      validator.update(lm.envImpl, entryBuffer, LogManager.HEADER_CONTENT_BYTES, false);
82      entryBuffer.position(oldpos);
83     }
84    }
85   }
86
87   after(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader, long lsn) throws
         DatabaseException : getRemainingBuffer(lm, entryBuffer, entryHeader, lsn) {
88    if (driver.isActivated("checksum")) {
89     if (lm.doChecksumOnRead) {
90      validator.update(lm.envImpl, entryBuffer, entryHeader.getEntrySize(), false);
91      validator.validate(lm.envImpl, entryHeader.getChecksum(), lsn);
92     }
93    }
94   }
95
96   before(EntryHeader eh, ByteBuffer dataBuffer): readHeader2(eh, dataBuffer) {
97    if (driver.isActivated("checksum")) {
98     eh.checksum = LogUtils.getUnsignedInt(dataBuffer);
99    }
```

```
100  }
101
102  after(FileReader fr) : lasFileReaderConstructor(fr) {
103   if (driver.isActivated("checksum")) {
104    fr.anticipateChecksumErrors = true;
105   }
106  }
107
108  after() returning (FileReader fr) : iNFileReaderConstructor() {
109   if (driver.isActivated("checksum")) {
110    fr.setAlwaysValidateChecksum(true);
111   }
112  }
113
114  after(ByteBuffer data, int recStartPos, int itemSize) : hook_recomputeChecksum(data,
         recStartPos,itemSize) {
115   if (driver.isActivated("checksum")) {
116    Checksum checksum = Adler32.makeChecksum();
117    data.position(recStartPos);
118    int nChecksumBytes = itemSize + (LogManager.HEADER_BYTES - LogManager.CHECKSUM_BYTES
           );
119    byte[] checksumBytes = new byte[nChecksumBytes];
120    System.arraycopy(data.array(), recStartPos + LogManager.CHECKSUM_BYTES,
           checksumBytes, 0, nChecksumBytes);
121    checksum.update(checksumBytes, 0, nChecksumBytes);
122    LogUtils.writeUnsignedInt(data, checksum.getValue());
123   }
124  }
125
126  boolean around() : readNextEntry() {
127   if (driver.isActivated("checksum")) {
128    boolean r = false;
129    try {
130     r = proceed();
131    } catch (DbChecksumException e) {
132    }
133    return r;
134   }
135   return proceed();
136  }
137
138  boolean around(RandomAccessFile newFile, String fileName, FileManager fm) throws
         DatabaseException: readAndValidateFileHeader(newFile, fileName, fm) {
139   if (driver.isActivated("checksum")) {
140    try {
141     return proceed(newFile, fileName, fm);
142    } catch (DbChecksumException e) {
143     fm.closeFileInErrorCase(newFile);
144     throw new DbChecksumException(fm.envImpl, "Couldn't␣open␣file␣" + fileName, e);
145    }
146   }
147   return proceed(newFile, fileName, fm);
148  }
149
```

```
150  before(ByteBuffer dataBuffer, FileReader fileReader) : readHeader3(dataBuffer,
         fileReader) {
151   if (driver.isActivated("checksum")) {
152    fileReader.currentEntryChecksum = LogUtils.getUnsignedInt(dataBuffer);
153   }
154  }
155
156  after(): staticinitialization(LogManager) {
157   if (driver.isActivated("checksum")) {
158    LogManager.HEADER_BYTES += LogManager.CHECKSUM_BYTES;
159    LogManager.PREV_BYTES = 4;
160    LogManager.HEADER_CONTENT_BYTES = LogManager.HEADER_BYTES - LogManager.
           CHECKSUM_BYTES;
161    LogManager.HEADER_ENTRY_TYPE_OFFSET += 4;
162    LogManager.HEADER_VERSION_OFFSET += 4;
163    LogManager.HEADER_PREV_OFFSET += 4;
164    LogManager.HEADER_SIZE_OFFSET += 4;
165   }
166  }
167
168  after()returning(ByteBuffer buffer) : allocate() {
169   if (driver.isActivated("checksum")) {
170    buffer.position(LogManager.CHECKSUM_BYTES);
171   }
172  }
173
174  after(StringBuffer sb, PrintFileReader pfr) : hook_printChecksum(sb, pfr) {
175   if (driver.isActivated("checksum")) {
176    sb.append("\"␣cksum=\"").append(pfr.currentEntryChecksum);
177   }
178  }
179 }
```

## A.2 Pointcut Redefinition

**Listing A.4** ChecksumDynamic.

```
1  package com.checksum;
2
3  import java.io.RandomAccessFile;
4  import java.nio.ByteBuffer;
5  import com.sleepycat.je.DatabaseException;
6  import com.sleepycat.je.cleaner.FileProcessor;
7  import com.sleepycat.je.dbi.EnvironmentImpl;
8  import com.sleepycat.je.log.EntryHeader;
9  import com.sleepycat.je.log.FileManager;
10 import com.sleepycat.je.log.FileReader;
11 import com.sleepycat.je.log.LogManager;
12 import com.sleepycat.je.log.PrintFileReader;
13 import com.sleepycat.je.log.entry.LogEntry;
14 import com.sleepycat.je.recovery.RecoveryManager;
```

```
15 import com.sleepycat.je.log.*;
16 import com.sleepycat.je.log.LogSource;
17
18 public privileged abstract aspect ChecksumAbstract {
19
20  pointcut fileReaderConstructor(FileReader fileReader, EnvironmentImpl env) : execution
        (FileReader.new(EnvironmentImpl, int, boolean, long, Long, long, long)) && args(
        env,int, boolean, long, Long, long,long) && this(fileReader) && within(FileReader)
        ;
21
22  pointcut hook_checksumValidation(FileReader fileReader, ByteBuffer dataBuffer) : call(
        void FileReader.hook_checksumValidation(ByteBuffer)) && target(fileReader) && args
        (dataBuffer);
23
24  pointcut hook_checkType(FileReader fr, byte currentEntryTypeNum) : execution(void
        FileReader.hook_checkType(byte)) && args(currentEntryTypeNum) && this(fr);
25
26  pointcut logManagerConstructor(LogManager logManager, EnvironmentImpl env) : execution
        (LogManager.new(EnvironmentImpl, boolean)) && this(logManager) && args(env,boolean
        );
27
28  pointcut addPrevOffset(int entrySize) : execution(ByteBuffer LogManager.addPrevOffset(
        ByteBuffer,long,int)) && args(ByteBuffer,long,entrySize) && within(LogManager);
29
30  pointcut readHeader(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader) :
        call(void EntryHeader.readHeader(ByteBuffer, boolean)) && this(lm) && args(
        entryBuffer,boolean) && target(entryHeader);
31
32  pointcut getRemainingBuffer(LogManager lm, ByteBuffer entryBuffer, EntryHeader
        entryHeader, long lsn) : call(ByteBuffer LogManager.getRemainingBuffer(ByteBuffer,
         LogSource, long, EntryHeader)) && target(lm) && args(entryBuffer, LogSource, long
        , entryHeader) && cflow(getLogEntryFromLogSource(lsn));
33
34  pointcut readHeader2(EntryHeader eh, ByteBuffer dataBuffer) : execution(void
        EntryHeader.readHeader(ByteBuffer,boolean)) && target(eh) && args(dataBuffer,
        boolean);
35
36  pointcut lasFileReaderConstructor(FileReader fr) : execution(LastFileReader.new(..))
       && this(fr);
37
38  pointcut iNFileReaderConstructor() : (call(INFileReader.new(..)) && withincode(void
        RecoveryManager.readINsAndTrackIds(long))) || (call(CleanerFileReader.new(..)) &&
        within(FileProcessor));
39
40  pointcut hook_recomputeChecksum(ByteBuffer data, int recStartPos, int itemSize) : call
        (void FileManager.hook_recomputeChecksum(ByteBuffer, int, int)) && args(data,
        recStartPos,itemSize);
41
42  pointcut readNextEntry() : execution(boolean LastFileReader.readNextEntry());
43
44  pointcut readAndValidateFileHeader(RandomAccessFile newFile, String fileName,
        FileManager fm) : execution(boolean FileManager.readAndValidateFileHeader(
        RandomAccessFile, String, long)) && args(newFile,fileName,long) && target(fm) &&
        within(FileManager);
```

```
45
46  pointcut readHeader3(ByteBuffer dataBuffer, FileReader fileReader) : execution(void
        FileReader.readHeader(ByteBuffer)) && args(dataBuffer) && target(fileReader);
47
48  pointcut allocate() : call(ByteBuffer ByteBuffer.allocate(int)) && withincode(
        ByteBuffer LogManager.marshallIntoBuffer(LoggableObject, int, boolean, int));
49
50  pointcut hook_printChecksum(StringBuffer sb, PrintFileReader pfr) : call(void
        PrintFileReader.hook_printChecksum(StringBuffer)) && args(sb) && this(pfr);
51
52  private pointcut getLogEntryFromLogSource(long lsn) : execution(LogEntry LogManager.
        getLogEntryFromLogSource(long,LogSource)) && args(lsn, LogSource);
53
54  Object around(FileReader fileReader, EnvironmentImpl env) throws DatabaseException :
        fileReaderConstructor(fileReader, env) {
55   fileReader.doValidateChecksum = env.getLogManager().getChecksumOnRead();
56   Object r = proceed(fileReader, env);
57   if (fileReader.doValidateChecksum) {
58    fileReader.cksumValidator = new ChecksumValidator();
59   }
60   fileReader.anticipateChecksumErrors = false;
61   return r;
62  }
63
64  after(FileReader fileReader, ByteBuffer dataBuffer) throws DatabaseException :
        hook_checksumValidation(fileReader, dataBuffer){
65   boolean doValidate = fileReader.doValidateChecksum && (fileReader.
         alwaysValidateChecksum || fileReader .isTargetEntry(fileReader.
         currentEntryTypeNum, fileReader.currentEntryTypeVersion));
66   if (doValidate) {
67    fileReader.startChecksum(dataBuffer);
68   }
69   if (doValidate)
70    fileReader.currentEntryCollectData = true;
71  }
72
73  before(FileReader fr, byte currentEntryTypeNum) throws DatabaseException :
        hook_checkType(fr, currentEntryTypeNum)     {
74   if (!LogEntryType.isValidType(currentEntryTypeNum))
75    throw new DbChecksumException((fr.anticipateChecksumErrors ? null : fr.env), "
         FileReader␣read␣invalid␣log␣entry␣type:␣" + currentEntryTypeNum);
76  }
77
78  after(LogManager logManager, EnvironmentImpl env) throws DatabaseException :
        logManagerConstructor(logManager, env) {
79   DbConfigManager configManager = env.getConfigManager();
80   logManager.doChecksumOnRead = configManager.getBoolean(EnvironmentParams.
        LOG_CHECKSUM_READ);
81  }
82
83  ByteBuffer around(int entrySize) : addPrevOffset(entrySize) {
84   ByteBuffer destBuffer = proceed(entrySize);
85   Checksum checksum = Adler32.makeChecksum();
```

```
86   checksum.update(destBuffer.array(), LogManager.CHECKSUM_BYTES, (entrySize -
         LogManager.CHECKSUM_BYTES));
87   LogUtils.writeUnsignedInt(destBuffer, checksum.getValue());
88   destBuffer.position(0);
89   return destBuffer;
90  }
91
92  after(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader) throws
        DatabaseException : readHeader(lm, entryBuffer, entryHeader) {
93   if (lm.doChecksumOnRead) {
94    validator = new ChecksumValidator();
95    int oldpos = entryBuffer.position();
96    entryBuffer.position(oldpos - LogManager.HEADER_CONTENT_BYTES);
97    validator.update(lm.envImpl, entryBuffer, LogManager.HEADER_CONTENT_BYTES, false);
98    entryBuffer.position(oldpos);
99   }
100 }
101
102 after(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader, long lsn) throws
        DatabaseException : getRemainingBuffer(lm, entryBuffer, entryHeader, lsn) {
103  if (lm.doChecksumOnRead) {
104   validator.update(lm.envImpl, entryBuffer, entryHeader.getEntrySize(), false);
105   validator.validate(lm.envImpl, entryHeader.getChecksum(), lsn);
106  }
107 }
108
109 before(EntryHeader eh, ByteBuffer dataBuffer): readHeader2(eh, dataBuffer) {
110  eh.checksum = LogUtils.getUnsignedInt(dataBuffer);
111 }
112
113 after(FileReader fr) : lasFileReaderConstructor(fr) {
114  fr.anticipateChecksumErrors = true;
115 }
116
117 after() returning (FileReader fr) : iNFileReaderConstructor() {
118  fr.setAlwaysValidateChecksum(true);
119 }
120
121 after(ByteBuffer data, int recStartPos, int itemSize) : hook_recomputeChecksum(data,
        recStartPos,itemSize) {
122  Checksum checksum = Adler32.makeChecksum();
123  data.position(recStartPos);
124  int nChecksumBytes = itemSize + (LogManager.HEADER_BYTES - LogManager.CHECKSUM_BYTES)
         ;
125  byte[] checksumBytes = new byte[nChecksumBytes];
126  System.arraycopy(data.array(), recStartPos + LogManager.CHECKSUM_BYTES, checksumBytes
         , 0, nChecksumBytes);
127  checksum.update(checksumBytes, 0, nChecksumBytes);
128  LogUtils.writeUnsignedInt(data, checksum.getValue());
129 }
130
131 boolean around() : readNextEntry() {
132  boolean r = false;
133  try {
```

```
134     r = proceed();
135    } catch (DbChecksumException e) {
136    }
137    return r;
138   }
139
140   boolean around(RandomAccessFile newFile, String fileName, FileManager fm) throws
          DatabaseException: readAndValidateFileHeader(newFile, fileName, fm) {
141    try {
142     return proceed(newFile, fileName, fm);
143    } catch (DbChecksumException e) {
144     fm.closeFileInErrorCase(newFile);
145     throw new DbChecksumException(fm.envImpl, "Couldn't␣open␣file␣" + fileName, e);
146    }
147   }
148
149   before(ByteBuffer dataBuffer, FileReader fileReader) : readHeader3(dataBuffer,
          fileReader) {
150    fileReader.currentEntryChecksum = LogUtils.getUnsignedInt(dataBuffer);
151   }
152
153   after(): staticinitialization(LogManager) {
154    LogManager.HEADER_BYTES += LogManager.CHECKSUM_BYTES;
155    LogManager.PREV_BYTES = 4;
156    LogManager.HEADER_CONTENT_BYTES = LogManager.HEADER_BYTES - LogManager.CHECKSUM_BYTES
           ;
157    LogManager.HEADER_ENTRY_TYPE_OFFSET += 4;
158    LogManager.HEADER_VERSION_OFFSET += 4;
159    LogManager.HEADER_PREV_OFFSET += 4;
160    LogManager.HEADER_SIZE_OFFSET += 4;
161   }
162
163   after()returning(ByteBuffer buffer) : allocate() {
164    buffer.position(LogManager.CHECKSUM_BYTES);
165   }
166
167   after(StringBuffer sb, PrintFileReader pfr) : hook_printChecksum(sb, pfr){
168    sb.append("\"␣cksum=\"").append(pfr.currentEntryChecksum);
169   }
170
171   public ChecksumValidator FileReader.cksumValidator;
172
173   private boolean FileReader.doValidateChecksum;
174
175   private boolean FileReader.alwaysValidateChecksum;
176
177   public boolean FileReader.anticipateChecksumErrors;
178
179   private void FileReader.startChecksum(ByteBuffer dataBuffer) throws DatabaseException
          {
180    cksumValidator.reset();
181    int entryStart = threadSafeBufferPosition(dataBuffer);
182    dataBuffer.reset();
```

```
183   cksumValidator.update(env, dataBuffer, LogManager.HEADER_CONTENT_BYTES,
          anticipateChecksumErrors);
184   threadSafeBufferPosition(dataBuffer, entryStart);
185  }
186
187  public long FileReader.currentEntryChecksum;
188
189  private void FileReader.validateChecksum(ByteBuffer entryBuffer) throws
         DatabaseException {
190
191   cksumValidator.update(env, entryBuffer, currentEntrySize, anticipateChecksumErrors);
192   cksumValidator.validate(env, currentEntryChecksum, readBufferFileNum,
          currentEntryOffset, anticipateChecksumErrors);
193  }
194
195  public void FileReader.setAlwaysValidateChecksum(boolean validate) {
196   alwaysValidateChecksum = validate;
197  }
198
199  private boolean LogManager.doChecksumOnRead;
200
201  public boolean LogManager.getChecksumOnRead() {
202   return doChecksumOnRead;
203  }
204
205  private ChecksumValidator validator = null;
206
207  private long EntryHeader.checksum;
208
209  public long EntryHeader.getChecksum() {
210   return checksum;
211  }
212
213  static final int LogManager.CHECKSUM_BYTES = 4;
214
215  static final int LogManager.HEADER_CHECKSUM_OFFSET = 0;
216 }
```

**Listing A.5** ChecksumStatic.

```
1 package com.checksum;
2
3 public privileged aspect ChecksumValidatorAspectStatic extends ChecksumAbstract {
4 }
```

**Listing A.6** ChecksumDynamic.

```
1 package com.checksum;
2
3 import java.io.RandomAccessFile;
4 import java.nio.ByteBuffer;
5 import com.sleepycat.je.dbi.EnvironmentImpl;
6 import com.sleepycat.je.log.EntryHeader;
```

```
7  import com.sleepycat.je.log.FileManager;
8  import com.sleepycat.je.log.FileReader;
9  import com.sleepycat.je.log.LogManager;
10 import com.sleepycat.je.log.PrintFileReader;
11
12 import driver.Driver;
13
14 public privileged aspect ChecksumValidatorAspect extends ChecksumAbstract {
15
16  pointcut driver() : if(new Driver().isActivated("checksum"));
17
18  pointcut fileReaderConstructor(FileReader fileReader, EnvironmentImpl env) :
        ChecksumAbstract.fileReaderConstructor(fileReader, env) && driver();
19
20  pointcut hook_checksumValidation(FileReader fileReader, ByteBuffer dataBuffer) :
        ChecksumAbstract.hook_checksumValidation(fileReader, dataBuffer) && driver();
21
22  pointcut hook_checkType(FileReader fr, byte currentEntryTypeNum) : ChecksumAbstract.
        hook_checkType(fr, currentEntryTypeNum) && driver();
23
24  pointcut logManagerConstructor(LogManager logManager, EnvironmentImpl env) :
        ChecksumAbstract.logManagerConstructor(logManager, env) && driver();
25
26  pointcut addPrevOffset(int entrySize) : ChecksumAbstract.addPrevOffset(entrySize) &&
        driver();
27
28  pointcut readHeader(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader) :
        ChecksumAbstract.readHeader(lm, entryBuffer, entryHeader) && driver();
29
30  pointcut getRemainingBuffer(LogManager lm, ByteBuffer entryBuffer, EntryHeader
        entryHeader, long lsn) : ChecksumAbstract.getRemainingBuffer(lm, entryBuffer,
        entryHeader, lsn) && driver();
31
32  pointcut readHeader2(EntryHeader eh, ByteBuffer dataBuffer) : ChecksumAbstract.
        readHeader2(eh, dataBuffer) && driver();
33
34  pointcut lasFileReaderConstructor(FileReader fr) : ChecksumAbstract.
        lasFileReaderConstructor(fr) && driver();
35
36  pointcut iNFileReaderConstructor() : ChecksumAbstract.iNFileReaderConstructor() &&
        driver();
37
38  pointcut hook_recomputeChecksum(ByteBuffer data, int recStartPos, int itemSize) :
        ChecksumAbstract.hook_recomputeChecksum(data, recStartPos, itemSize) && driver();
39
40  pointcut readNextEntry() : ChecksumAbstract.readNextEntry() && driver();
41
42  pointcut readAndValidateFileHeader(RandomAccessFile newFile, String fileName,
        FileManager fm) : ChecksumAbstract.readAndValidateFileHeader(newFile, fileName, fm
        ) && driver();
43
44  pointcut readHeader3(ByteBuffer dataBuffer, FileReader fileReader) : ChecksumAbstract.
        readHeader3(dataBuffer, fileReader) && driver();
45
```

```
46  pointcut allocate() : ChecksumAbstract.allocate() && driver();
47
48  pointcut hook_printChecksum(StringBuffer sb, PrintFileReader pfr) : ChecksumAbstract.
        hook_printChecksum(sb, pfr) && driver();
49
50  pointcut getLogEntryFromLogSource(long lsn) : ChecksumAbstract.
        getLogEntryFromLogSource(lsn) && driver();
51 }
```

## A.3 Layered Aspects

The `ChecksumAbstract` and the `ChecksumStatic` aspects are equal to the one in Listing A.4 and A.5, respectively.

**Listing A.7** ChecksumDynamic.

```
1  package com.checksum;
2
3  import java.io.RandomAccessFile;
4
5  import com.sleepycat.je.dbi.EnvironmentImpl;
6  import com.sleepycat.je.log.FileManager;
7  import com.sleepycat.je.log.FileReader;
8
9  import driver.Driver;
10
11 public privileged aspect ChecksumValidatorAspect extends ChecksumAbstract {
12
13  pointcut driver() : if(new Driver().isActivated("checksum"));
14
15  pointcut fileReaderConstructor(FileReader fileReader, EnvironmentImpl env) :
        ChecksumAbstract.fileReaderConstructor(fileReader, env) && driver();
16
17  pointcut addPrevOffset(int entrySize) : ChecksumAbstract.addPrevOffset(entrySize) &&
        driver();
18
19  pointcut readNextEntry() : ChecksumAbstract.readNextEntry() && driver();
20
21  pointcut readAndValidateFileHeader(RandomAccessFile newFile, String fileName,
        FileManager fm) : ChecksumAbstract.readAndValidateFileHeader(newFile, fileName, fm
        ) && driver();
22
23  Object around() : adviceexecution() && within(com.checksum.ChecksumAbstract) && !
        driver() {
24   return null;
25  }
26 }
```

## A.4 Flexible Deployment

**Listing A.8** ChecksumDynamic.

```
1  package com.checksum;
2
3  import java.awt.List;
4  import java.io.RandomAccessFile;
5  import java.nio.ByteBuffer;
6  import java.util.ArrayList;
7  import java.util.zip.Checksum;
8  import com.sleepycat.je.DatabaseException;
9  import com.sleepycat.je.dbi.EnvironmentImpl;
10 import com.sleepycat.je.log.EntryHeader;
11 import com.sleepycat.je.log.FileManager;
12 import com.sleepycat.je.log.FileReader;
13 import com.sleepycat.je.log.LogManager;
14 import com.sleepycat.je.log.PrintFileReader;
15 import com.sleepycat.je.config.EnvironmentParams;
16 import com.sleepycat.je.dbi.DbConfigManager;
17 import com.sleepycat.je.log.LogEntryType;
18 import com.sleepycat.je.log.LogUtils;
19 import com.sleepycat.je.cleaner.FileProcessor;
20 import com.sleepycat.je.log.entry.LogEntry;
21 import com.sleepycat.je.recovery.RecoveryManager;
22 import com.sleepycat.je.log.*;
23
24 public cclass ChecksumAbstract {
25
26  public static final ChecksumAbstract instance = new ChecksumAbstract();
27
28  pointcut fileReaderConstructor(FileReader fileReader, EnvironmentImpl env) : execution
       (FileReader.new(EnvironmentImpl, int, boolean, long, Long, long, long)) && args(
       env,int, boolean, long, Long, long,long) && this(fileReader) && within(FileReader)
       ;
29
30  pointcut hook_checksumValidation(FileReader fileReader, ByteBuffer dataBuffer) : call(
       void FileReader.hook_checksumValidation(ByteBuffer)) && target(fileReader) && args
       (dataBuffer);
31
32  pointcut hook_checkType(FileReader fr, byte currentEntryTypeNum) : execution(void
       FileReader.hook_checkType(byte)) && args(currentEntryTypeNum) && this(fr);
33
34  pointcut logManagerConstructor(LogManager logManager, EnvironmentImpl env) : execution
       (LogManager.new(EnvironmentImpl, boolean)) && this(logManager) && args(env,boolean
       );
35
36  pointcut addPrevOffset(int entrySize) : execution(ByteBuffer LogManager.addPrevOffset(
       ByteBuffer,long,int)) && args(ByteBuffer,long,entrySize) && within(LogManager);
37
38  pointcut readHeader(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader) :
       call(void EntryHeader.readHeader(ByteBuffer, boolean)) && this(lm) && args(
       entryBuffer,boolean) && target(entryHeader);
39
40  pointcut getRemainingBuffer(LogManager lm, ByteBuffer entryBuffer, EntryHeader
       entryHeader, long lsn) : call(ByteBuffer LogManager.getRemainingBuffer(ByteBuffer,
```

```
      LogSource, long, EntryHeader)) && target(lm) && args(entryBuffer, LogSource, long
      , entryHeader) && cflow(getLogEntryFromLogSource(lsn));
41
42  pointcut readHeader2(EntryHeader eh, ByteBuffer dataBuffer) : execution(void
      EntryHeader.readHeader(ByteBuffer,boolean)) && target(eh) && args(dataBuffer,
      boolean);
43
44  pointcut lasFileReaderConstructor(FileReader fr) : execution(LastFileReader.new(..))
      && this(fr);
45
46  pointcut iNFileReaderConstructor() : (call(INFileReader.new(..)) && withincode(void
      RecoveryManager.readINsAndTrackIds(long))) || (call(CleanerFileReader.new(..)) &&
      within(FileProcessor));
47
48  pointcut hook_recomputeChecksum(ByteBuffer data, int recStartPos, int itemSize) : call
      (void FileManager.hook_recomputeChecksum(ByteBuffer, int, int)) && args(data,
      recStartPos,itemSize);
49
50  pointcut readNextEntry() : execution(boolean LastFileReader.readNextEntry());
51
52  pointcut readAndValidateFileHeader(RandomAccessFile newFile, String fileName,
      FileManager fm) : execution(boolean FileManager.readAndValidateFileHeader(
      RandomAccessFile, String, long)) && args(newFile,fileName,long) && target(fm) &&
      within(FileManager);
53
54  pointcut readHeader3(ByteBuffer dataBuffer, FileReader fileReader) : execution(void
      FileReader.readHeader(ByteBuffer)) && args(dataBuffer) && target(fileReader);
55
56  pointcut allocate() : call(ByteBuffer ByteBuffer.allocate(int)) && withincode(
      ByteBuffer LogManager.marshallIntoBuffer(LoggableObject, int, boolean, int));
57
58  pointcut hook_printChecksum(StringBuffer sb, PrintFileReader pfr) : call(void
      PrintFileReader.hook_printChecksum(StringBuffer)) && args(sb) && this(pfr);
59
60  private pointcut getLogEntryFromLogSource(long lsn) : execution(LogEntry LogManager.
      getLogEntryFromLogSource(long,LogSource)) && args(lsn, LogSource);
61
62  Object around(FileReader fileReader, EnvironmentImpl env) throws DatabaseException :
      fileReaderConstructor(fileReader, env) {
63   fileReader.doValidateChecksum = env.getLogManager().getChecksumOnRead();
64   Object r = proceed(fileReader, env);
65   if (fileReader.doValidateChecksum) {
66    fileReader.cksumValidator = new ChecksumValidator();
67   }
68   fileReader.anticipateChecksumErrors = false;
69   return r;
70  }
71
72  after(FileReader fileReader, ByteBuffer dataBuffer) throws DatabaseException :
      hook_checksumValidation(fileReader, dataBuffer){
73   boolean doValidate = fileReader.doValidateChecksum && (fileReader.
       alwaysValidateChecksum || fileReader .isTargetEntry(fileReader.
       currentEntryTypeNum, fileReader.currentEntryTypeVersion));
74   if (doValidate) {
```

```
75      fileReader.startChecksum(dataBuffer);
76    }
77    if (doValidate)
78      fileReader.currentEntryCollectData = true;
79  }
80
81  before(FileReader fr, byte currentEntryTypeNum) throws DatabaseException :
        hook_checkType(fr, currentEntryTypeNum)      {
82    if (!LogEntryType.isValidType(currentEntryTypeNum))
83      throw new DbChecksumException((fr.anticipateChecksumErrors ? null : fr.env), "
          FileReader␣read␣invalid␣log␣entry␣type:␣" + currentEntryTypeNum);
84  }
85
86  after(LogManager logManager, EnvironmentImpl env) throws DatabaseException :
        logManagerConstructor(logManager, env) {
87    DbConfigManager configManager = env.getConfigManager();
88    logManager.doChecksumOnRead = configManager.getBoolean(EnvironmentParams.
        LOG_CHECKSUM_READ);
89  }
90
91  ByteBuffer around(int entrySize) : addPrevOffset(entrySize) {
92    ByteBuffer destBuffer = proceed(entrySize);
93    Checksum checksum = Adler32.makeChecksum();
94    checksum.update(destBuffer.array(), LogManager.CHECKSUM_BYTES, (entrySize -
        LogManager.CHECKSUM_BYTES));
95    LogUtils.writeUnsignedInt(destBuffer, checksum.getValue());
96    destBuffer.position(0);
97    return destBuffer;
98  }
99
100 after(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader) throws
        DatabaseException : readHeader(lm, entryBuffer, entryHeader) {
101   if (lm.doChecksumOnRead) {
102     validator = new ChecksumValidator();
103     int oldpos = entryBuffer.position();
104     entryBuffer.position(oldpos - LogManager.HEADER_CONTENT_BYTES);
105     validator.update(lm.envImpl, entryBuffer, LogManager.HEADER_CONTENT_BYTES, false);
106     entryBuffer.position(oldpos);
107   }
108 }
109
110 after(LogManager lm, ByteBuffer entryBuffer, EntryHeader entryHeader, long lsn) throws
        DatabaseException : getRemainingBuffer(lm, entryBuffer, entryHeader, lsn) {
111   if (lm.doChecksumOnRead) {
112     validator.update(lm.envImpl, entryBuffer, entryHeader.getEntrySize(), false);
113     validator.validate(lm.envImpl, entryHeader.getChecksum(), lsn);
114   }
115 }
116
117 before(EntryHeader eh, ByteBuffer dataBuffer): readHeader2(eh, dataBuffer) {
118   eh.checksum = LogUtils.getUnsignedInt(dataBuffer);
119 }
120
121 after(FileReader fr) : lasFileReaderConstructor(fr) {
```

```
122   fr.anticipateChecksumErrors = true;
123  }
124
125  after() returning (FileReader fr) : iNFileReaderConstructor() {
126   fr.setAlwaysValidateChecksum(true);
127  }
128
129  after(ByteBuffer data, int recStartPos, int itemSize) : hook_recomputeChecksum(data,
         recStartPos,itemSize) {
130   Checksum checksum = Adler32.makeChecksum();
131   data.position(recStartPos);
132   int nChecksumBytes = itemSize + (LogManager.HEADER_BYTES - LogManager.CHECKSUM_BYTES)
          ;
133   byte[] checksumBytes = new byte[nChecksumBytes];
134   System.arraycopy(data.array(), recStartPos + LogManager.CHECKSUM_BYTES, checksumBytes
          , 0, nChecksumBytes);
135   checksum.update(checksumBytes, 0, nChecksumBytes);
136   LogUtils.writeUnsignedInt(data, checksum.getValue());
137  }
138
139  boolean around() : readNextEntry() {
140   boolean r = false;
141   try {
142    r = proceed();
143   } catch (DbChecksumException e) {
144   }
145   return r;
146  }
147
148  boolean around(RandomAccessFile newFile, String fileName, FileManager fm) throws
         DatabaseException: readAndValidateFileHeader(newFile, fileName, fm) {
149   try {
150    return proceed(newFile, fileName, fm);
151   } catch (DbChecksumException e) {
152    fm.closeFileInErrorCase(newFile);
153    throw new DbChecksumException(fm.envImpl, "Couldn't␣open␣file␣" + fileName, e);
154   }
155  }
156
157  before(ByteBuffer dataBuffer, FileReader fileReader) : readHeader3(dataBuffer,
         fileReader) {
158   fileReader.currentEntryChecksum = LogUtils.getUnsignedInt(dataBuffer);
159  }
160
161  after(): staticinitialization(LogManager) {
162   LogManager.HEADER_BYTES += LogManager.CHECKSUM_BYTES;
163   LogManager.PREV_BYTES = 4;
164   LogManager.HEADER_CONTENT_BYTES = LogManager.HEADER_BYTES - LogManager.CHECKSUM_BYTES
          ;
165   LogManager.HEADER_ENTRY_TYPE_OFFSET += 4;
166   LogManager.HEADER_VERSION_OFFSET += 4;
167   LogManager.HEADER_PREV_OFFSET += 4;
168   LogManager.HEADER_SIZE_OFFSET += 4;
169  }
```

```
170
171  after()returning(ByteBuffer buffer) : allocate() {
172   buffer.position(LogManager.CHECKSUM_BYTES);
173  }
174
175  after(StringBuffer sb, PrintFileReader pfr) : hook_printChecksum(sb, pfr){
176   sb.append("\" cksum=\"").append(pfr.currentEntryChecksum);
177  }
178
179  public cclass FileReaderCaesarJ wraps FileReader {
180   public ChecksumValidator cksumValidator;
181   public boolean doValidateChecksum;
182   public boolean alwaysValidateChecksum;
183   public boolean anticipateChecksumErrors;
184
185   public void setAnticipateChecksumErrors(boolean anticipateChecksumErrors) {
186    this.anticipateChecksumErrors = anticipateChecksumErrors;
187   }
188
189   public void setCksumValidator(ChecksumValidator cksumValidator) {
190    this.cksumValidator = cksumValidator;
191   }
192
193   public void setDoValidateChecksum(boolean doValidateChecksum) {
194    this.doValidateChecksum = doValidateChecksum;
195   }
196
197   public void startChecksum(ByteBuffer dataBuffer) throws DatabaseException {
198    cksumValidator.reset();
199    int entryStart = wrappee.threadSafeBufferPosition(dataBuffer);
200    dataBuffer.reset();
201    cksumValidator.update(wrappee.env, dataBuffer, LogManager.HEADER_CONTENT_BYTES,
          anticipateChecksumErrors);
202    wrappee.threadSafeBufferPosition(dataBuffer, entryStart);
203   }
204
205   public long currentEntryChecksum;
206
207   public void setCurrentEntryChecksum(long currentEntryChecksum) {
208    this.currentEntryChecksum = currentEntryChecksum;
209   }
210
211   private void validateChecksum(ByteBuffer entryBuffer) throws DatabaseException {
212    cksumValidator.update(wrappee.env, entryBuffer, wrappee.currentEntrySize,
          anticipateChecksumErrors);
213    cksumValidator.validate(wrappee.env, currentEntryChecksum, wrappee.readBufferFileNum
          , wrappee.currentEntryOffset, anticipateChecksumErrors);
214   }
215
216   public void setAlwaysValidateChecksum(boolean validate) {
217    alwaysValidateChecksum = validate;
218   }
219
220  }
```

```
221
222  public cclass LogManagerCaesarJ wraps LogManager {
223   private boolean doChecksumOnRead;
224   public boolean getChecksumOnRead() {
225    return doChecksumOnRead;
226   }
227
228   public void setDoChecksumOnRead(boolean doChecksumOnRead) {
229    this.doChecksumOnRead = doChecksumOnRead;
230   }
231
232   public static final int CHECKSUM_BYTES = 4;
233
234   public static final int HEADER_CHECKSUM_OFFSET = 0;
235  }
236
237  public cclass EntryHeaderCaesarJ wraps EntryHeader {
238   private long checksum;
239   public long getChecksum() {
240    return checksum;
241   }
242
243   public void setChecksum(long checksum) {
244    this.checksum = checksum;
245   }
246  }
247
248  private ChecksumValidator validator = null;
249 }
```

**Listing A.9** ChecksumStatic.

```
1 package com.checksum;
2
3 public deployed cclass ChecksumValidatorAspectStatic extends ChecksumAbstract {
4 }
```

**Listing A.10** ChecksumDynamic.

```
1  package com.checksum;
2
3  import com.sleepycat.je.utilint.JarMain;
4  import driver.Driver;
5
6  public deployed cclass ChecksumValidatorAspect {
7
8   pointcut pc_jarmain() : execution(* JarMain.main(..));
9
10  before() : pc_jarmain() {
11   if (new Driver().isActivated("checksum")) {
12    ChecksumAbstract checksumAbstract = new ChecksumAbstract();
13    deploy checksumAbstract;
14   }
```

```
15    }
16  }
```

# B

# Appendix B - Metric Results

In this appendix, we present the complete metric results of our evaluation. We provide for each metric, a graphic, which contains all the features from our four case studies.

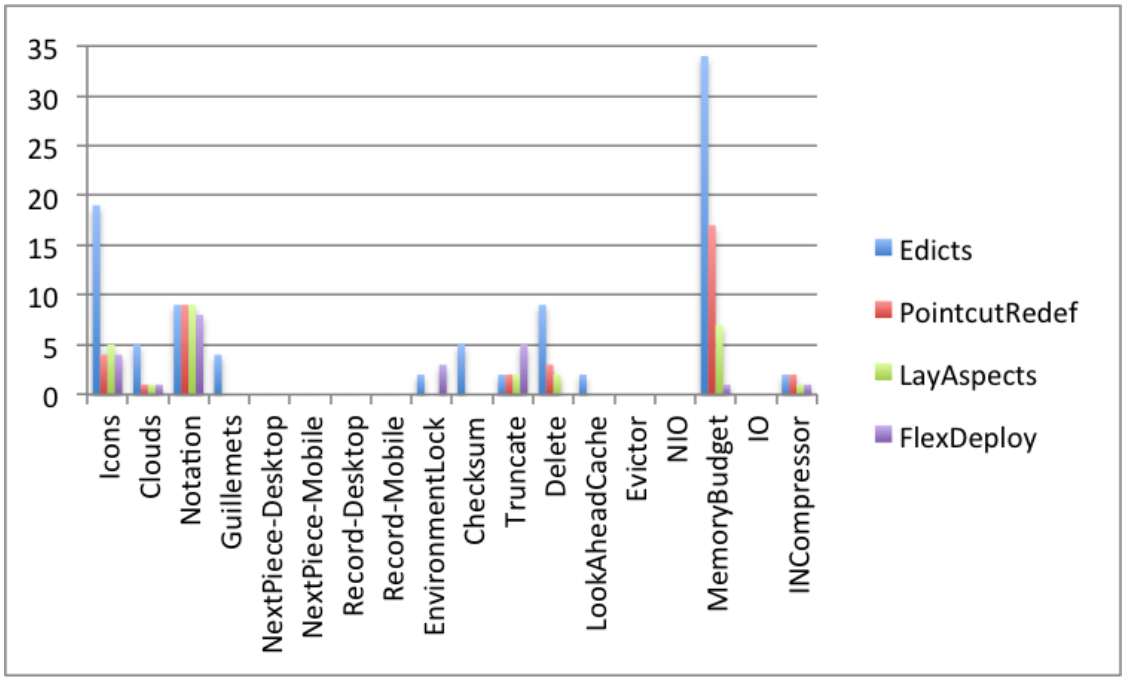



**Figure B.1** PCC metric results

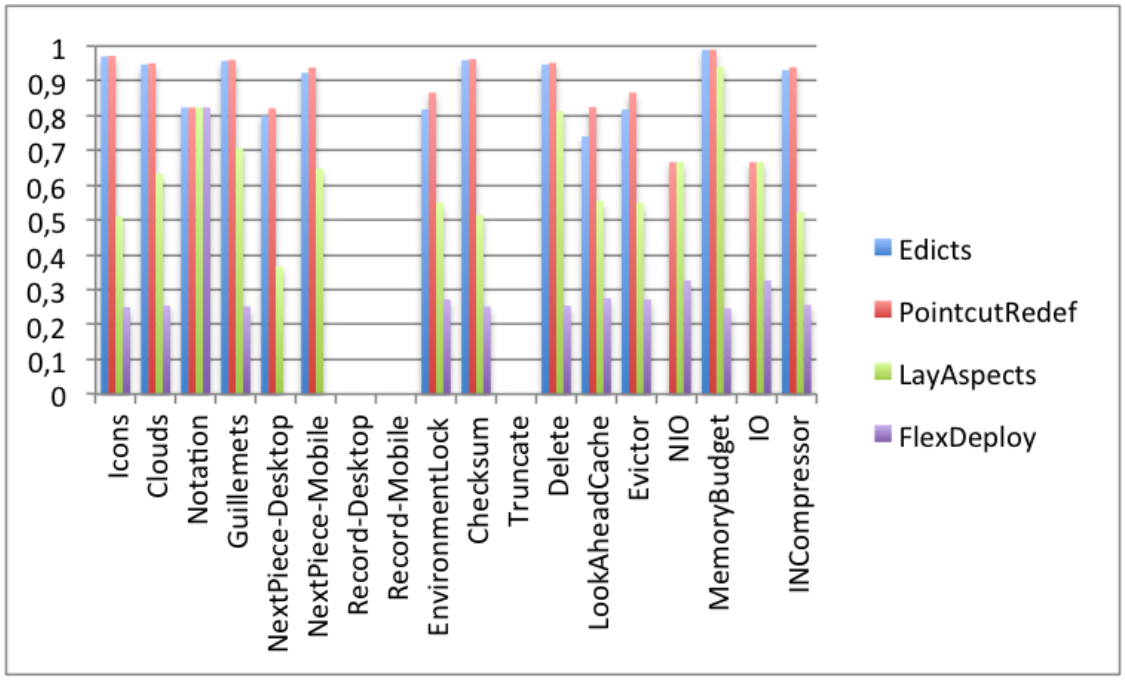



**Figure B.2** DOSO Driver metric results

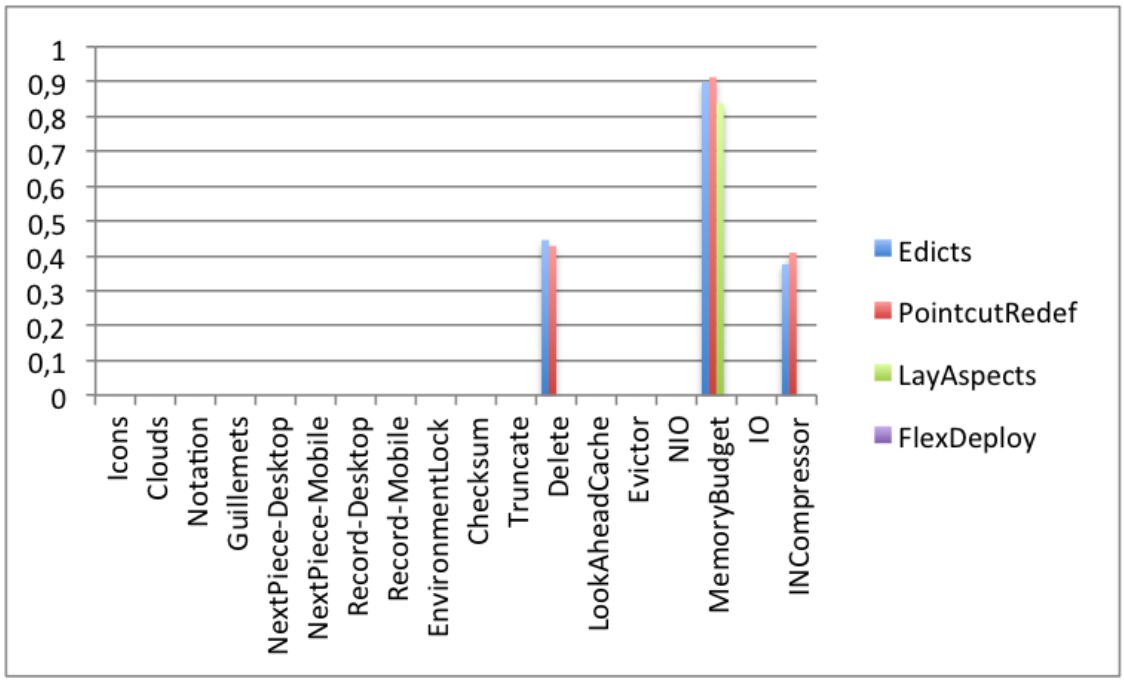



**Figure B.3** DOSC Driver metric results

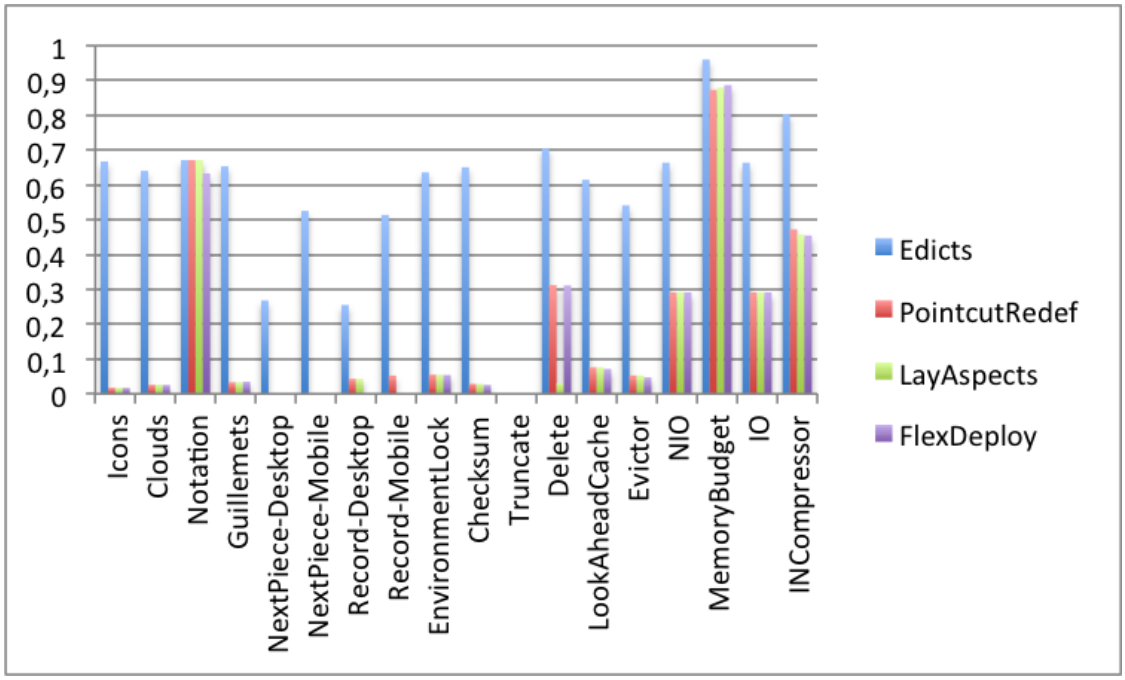



**Figure B.4** DOSC Feature metric results

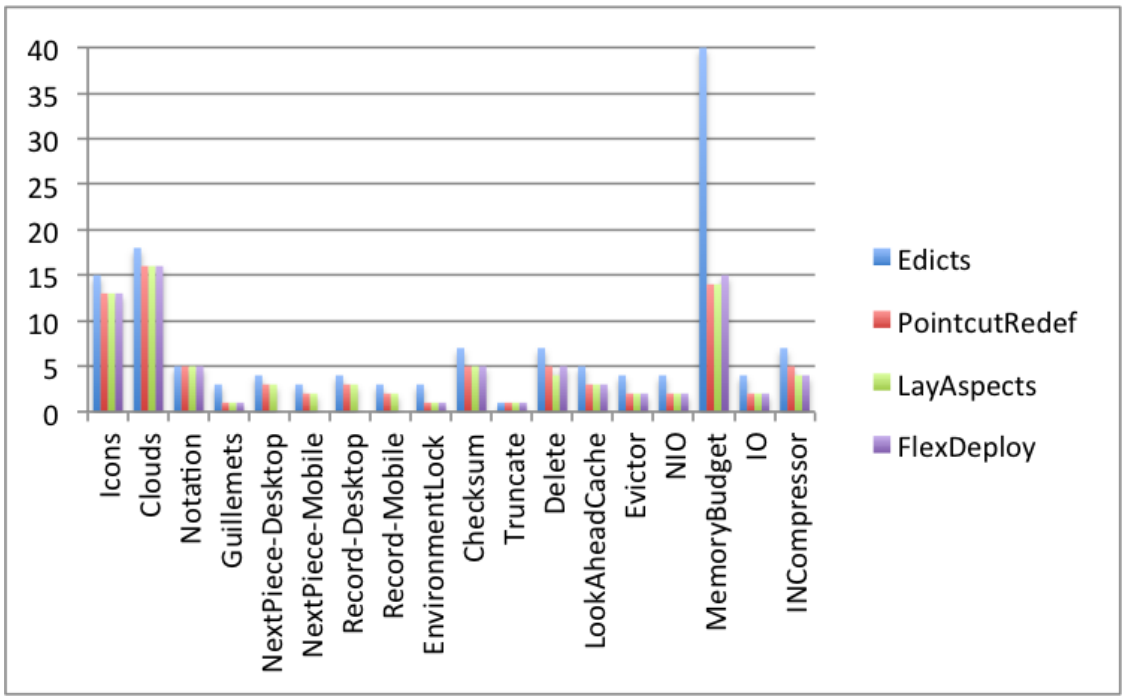



**Figure B.5** CDC metric results

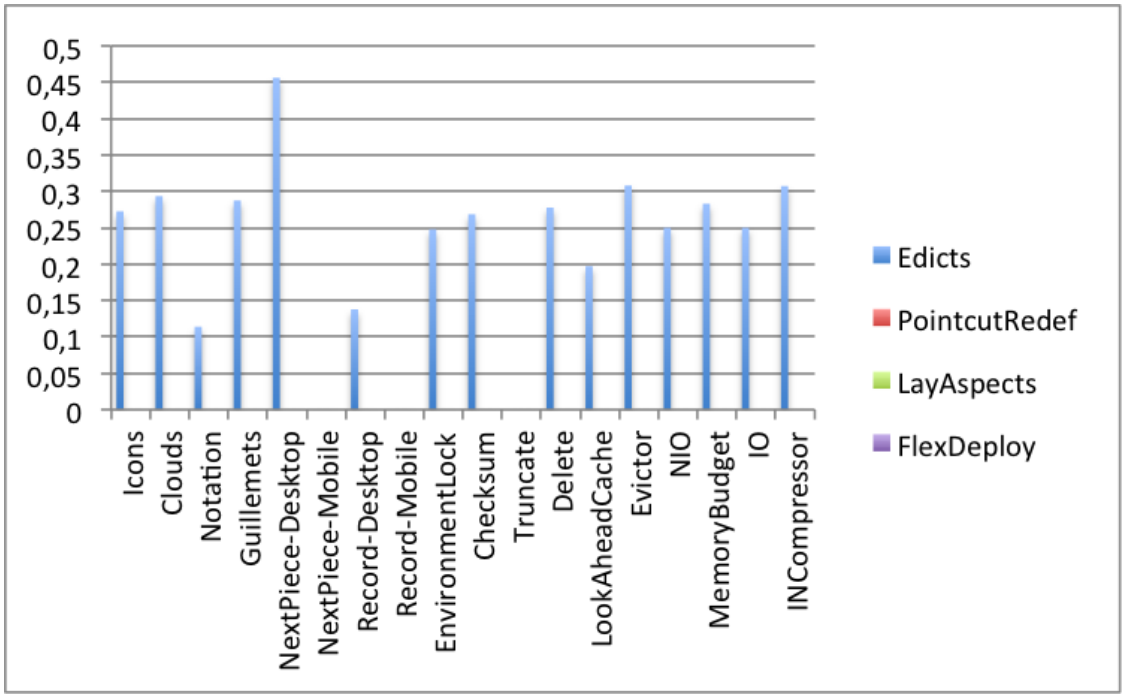



**Figure B.6** DOTO metric results

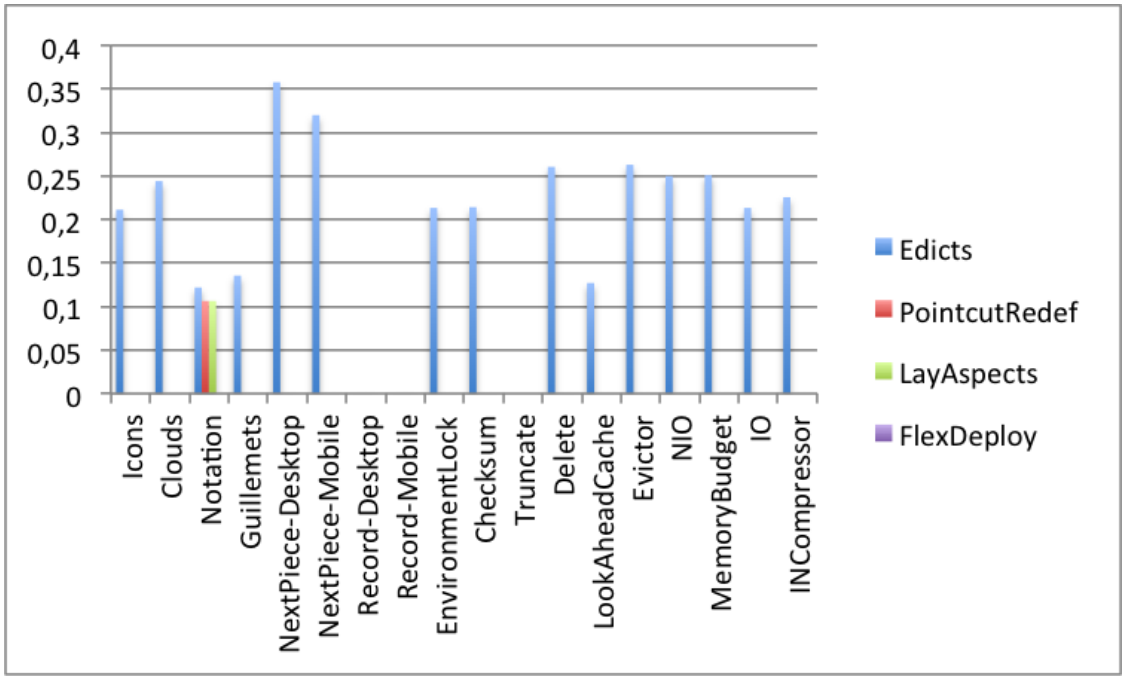



**Figure B.7** DOTC metric results

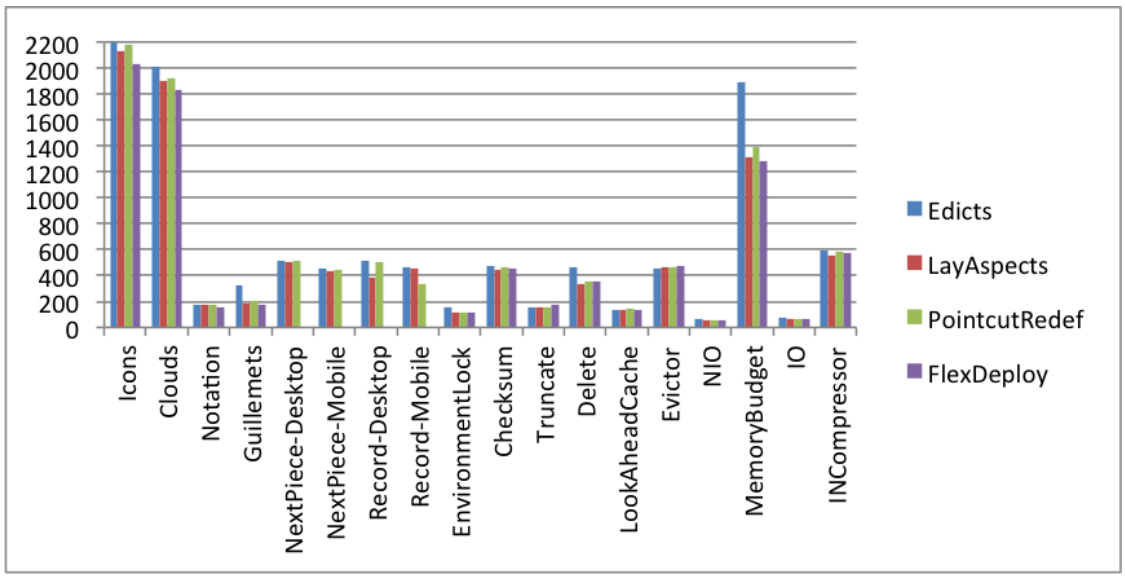



**Figure B.8** SLOC metric results

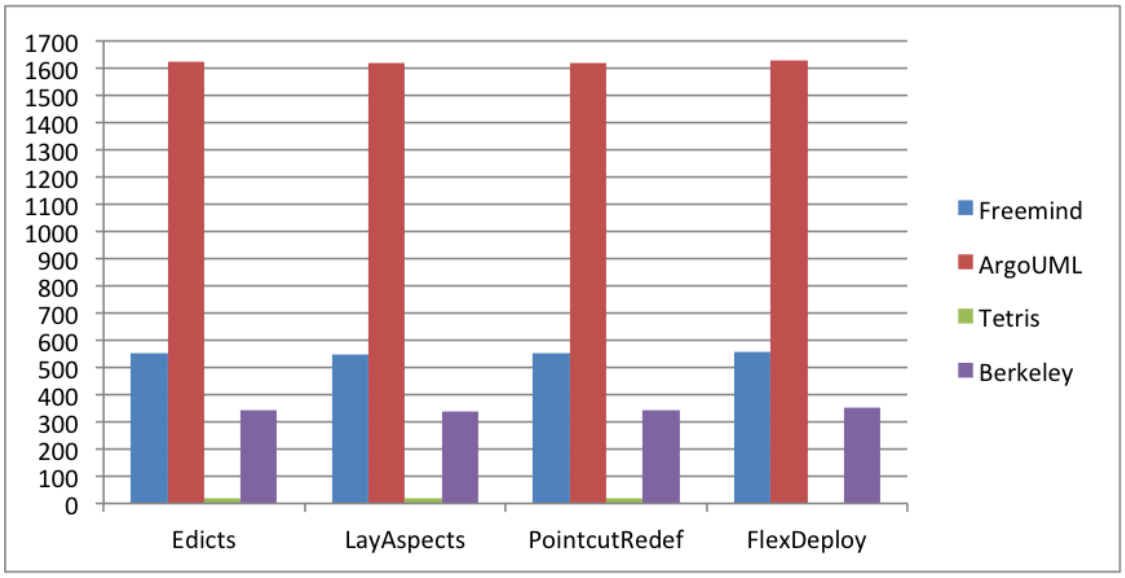



**Figure B.9** VS metric results