Assessing Idioms for Implementing Features with Flexible Binding Times

Rodrigo Andrade
Informatics Center
Federal University of Pernambuco
Recife, Brazil
rcaa2@cin.ufpe.br

Márcio Ribeiro
Informatics Center
Federal University of Pernambuco
Recife, Brazil
mmr3@cin.ufpe.br

Vaidas Gasiunas
Technische Universität Darmstadt
Darmstadt, Germany
gasiunas@st.informatik.tu-darmstadt.de

Lucas Satabin
Technische Universität Darmstadt
Darmstadt, Germany
satabin@st.informatik.tu-darmstadt.de

Henrique Rebêlo
Informatics Center
Federal University of Pernambuco
Recife, Brazil
hemr@cin.ufpe.br

Paulo Borba
Informatics Center
Federal University of Pernambuco
Recife, Brazil
phmb@cin.ufpe.br

***Abstract*—Maintainability of a software product line depends on the possibility to modularize its variations, often expressed in terms of optionally selected features. Conventional modularization techniques bind variations either statically or dynamically, but ideally it should be possible to flexibly choose between both. In this paper, we propose improved solutions for modularizing and flexibly binding varying features in form of idioms in aspect-oriented languages AspectJ and CaesarJ. We evaluate the idioms qualitatively by discussing their advantages and deficiencies and quantitatively by means of metrics.**

***Index Terms*—Flexible binding time; Modularity; Metrics; AspectJ; CaesarJ; Software Product Line; Aspects**

I. Introduction

Companies are adopting the Software Product Line (SPL) development paradigm in order 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 products by combining various features according to the requirements of specific customers [1].

Due to specific client requirements, it is important to define whether certain features should be activated or not. In this context, the binding time of a feature is the time at which one decides to activate or deactivate the feature from a product [2]. In general, two binding times are considered: 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 [3]. For systems without constrained resources, the binding time can be flexible, features can be added or removed statically or users may enable or disable a feature on demand (dynamically).

In this scenario, an idiom based on AspectJ [4] named Edicts was introduced with the goal to support the implementation of features with flexible binding times in a modular and convenient way [2]. Although the authors claim that Edicts is modular, no assessment was performed with respect to modularity. Actually, the situation gets even worse: we observe problems when using the Edicts idiom, such as code duplication. This way, the task of maintaining the system becomes time consuming and error-prone. In this work, we use Edicts to implement flexible binding time into several features from four systems (Tetris J2ME[1], Freemind [5], ArgoUML [6] and BerkeleyDB [7]).

In order to address the shortcomings of the Edicts idiom, we propose two alternative idioms developed by us for implementing flexible binding time: another AspectJ-based idiom that relies on *adviceexecution* pointcut and aspect inheritance, and an idiom based on the flexible deployment supported by the CaesarJ programming language [8]. For the purpose of evaluating our idioms and compare them with Edits, we also use both to implement flexible binding time for the same set of features and provide a quantitative evaluation of the resulting implementations.

In Sections III-A and III-B we introduce two novel idioms aiming at improving flexible binding time support for feature variability. Additionally, in Sections V-A V-B V-C and V-D, we present a quantitative evaluation of all tree idioms, based on several case studies, with regard to size, code repetition, scattering, and tangling. Furthermore, we show that some problems that directly affect the maintenance effort can be encountered in all of the idioms. Finally, in Section V-E, we discuss the idioms qualitatively.

II. Motivating Example

The same feature of a product line may need to have different binding time. For instance, the Facebook Farmville game needs to "dynamically turn off calls back to the platform" at peak times "since performance can be variable" [9]. Users would not be able to play Farmville at peak time

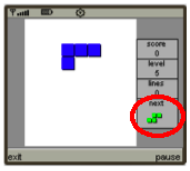
[1] http://kiang.org/jordan/software/tetrismidlet/

1534-5351/11 $26.00 © 2011 IEEE
DOI 10.1109/CSMR.2011.29

without disabling the call back feature dynamically due to its enormous amount of traffic. On the other hand, it is not necessary to disable the aforementioned feature for local networks, for example, when considering enterprise social network environment, so the game can send calls back to the platform at any time. Thus, we suggest dynamic binding time for the first and static binding time (compile-time) for the latter.

To implement flexible binding time, we could use the Edicts idiom [2], which modularizes SPL features and supports their flexible binding time. The idiom suggests modularizing features by means of AspectJ [4] aspects. For each feature, two concrete aspects are defined – one implementing static binding and another implementing dynamic binding of the feature. Both aspects inherit from an abstract aspect containing the common part of their functionality.

To illustrate how this works, consider the next-piece-box feature from Tetris J2ME. This feature is responsible for showing a box with the next piece that the game is about to drop on the screen. Such feature could be bound statically and be always present in the game (Fig. 1(a)), but also dynamically by providing the user with a dynamic option to switch this feature on or off (Fig. 1(b)).



(a) Static next-piece box.

(b) Dynamic next-piece box.

Fig. 1. Two instances of the Tetris game with different binding times for the next-piece-box feature.

Following the edicts idiom, we define two concrete aspects – one for the static and one for the dynamic binding of the next-piece-box feature – and one abstract aspect, which contains their shared functionality.

Aspect *EdictsNextPieceDynamic* (Listing 2) implements static binding by a set of pointcut-advice pairs that insert the functionality of the feature at corresponding places in the application: initialization of the next-piece box after initialization of the canvas (lines 6-10), and repainting of the box after repainting the canvas (lines 12-20). Each piece of advice must check whether the feature is currently turned on or not. For this purpose, we include if-statements that check whether the user choice is "Yes" (lines 7 and 14). The poincuts and the field storing the next-piece box are defined[2] in the abstract aspect *EdictsNextPieceAbstract* (Listing 1), which makes it possible to reuse them for the implementation of the dynamic binding, implemented by aspect *EdictsNextPieceDynamic* (Listing 2). The problem is that considering the static binding time, we need to create another aspect *EdictsNextPieceStatic* with the same feature code but without the *if* statements, since there is no user decision in that case (Listing 3).

Listing 1. EdictsNextPieceAbstract aspect

```
1: public abstract aspect EdictsNextPieceAbstract {
2:
3:   public NextPieceBox TetrisCanvas.nextPieceBox;
4:
5:   pointcut nextPiece(TetrisCanvas canvas):
6:     execution(private void TetrisCanvas.setupLayout(..))
7:     && this(canvas);
8:
9:   pointcut infoBoxes(Graphics g, TetrisCanvas canvas):
10:    execution(void TetrisCanvas.paintInfoBoxes(Graphics))
11:    && args(g) && this(canvas);
10:}
```

Listing 2. EdictsNextPieceDynamic aspect

```
1: public privileged aspect EdictsNextPieceDynamic
2:  extends EdictsNextPieceAbstract {
3:
4:  protected NextPieceBox nextPieceBox;
5:
6:  after(TetrisCanvas canvas): nextPiece(canvas) {
7:    if (userChoice.equals("Yes")) {
8:      nextPieceBox = new NextPieceBox(...);
9:    }
10: }
11:
12: after(Graphics g, TetrisCanvas canvas)
13:   infoBoxes(g, canvas) {
14:   if (userChoice.equals("Yes")) {
15:     if (nextPieceBox.setPieceType(
16:       canvas.game.getNextPieceType())) {
17:       nextPieceBox.paint(g);
18:     }
19:   }
20: }
21: ...
22:}
```

Listing 3. EdictsNextPieceStatic aspect

```
1: public privileged aspect EdictsNextPieceStatic
2:  extends EdictsNextPieceAbstract {
3:
4:    after(TetrisCanvas canvas): nextPiece(canvas) {
5:      nextPieceBox = new NextPieceBox(...);
6:        }
7:
8:    after(Graphics g, TetrisCanvas canvas)
9:     :infoBoxes(g, canvas) {
10:     if (canvas.nextPieceBox.setPieceType(
11:       canvas.game.getNextPieceType())) {
12:       nextPieceBox.paint(g);
13:         }
14:       }
15:  ...
16: }
```

If we need to have the next piece box feature always turned on, we must use static binding time, so it is necessary to include the *EdictsNextPieceStatic* and *EdictsNextPieceAbstract* into the product. Otherwise, we can let the user decide it, in

[2]AspectJ's intertype declarations allow adding new members and inheritance relationships to existing classes, e.g., the declaration on Line 3 adds field *nextPieceBox* to class *TetrisCanvas*.

this case, we use dynamic binding time, so it is required to include the *EdictsNextPieceDynamic* and *EdictsNextPieceAbstract*. We separate static and dynamic contexts because of the performance overhead introduced by the latter [3].

The problem with the presented solution is that there is a significant amount of code duplication among the static and dynamic aspects. In particular we must duplicate declarations of all pieces of advice, because in case of the dynamic binding, their implementations must be additionally wrapped into an if-statement implementing the dynamic check.[3] Second, the dynamic check determining activation of the feature is repeated for each piece of advice and scattered with the implementation of the dynamic aspect.

Code duplication is a disadvantage for maintainability, because certain changes to the code must be repeated. It is also more error-prone, because the developer may forget to update the repeated code consitently. Updating dynamic conditions for activation of a feature may be especially problematic. For instance, we may decide to make the game smarter so that it activates the next-piece feature automatically, when a player is about to lose a game. This condition has a completely different implementation when compared to the configuration by a manual option showed in Fig. 1(b). We must add the new condition repeatedly for all the advice that implement feature code. However, we suggest that this implementation does not fit properly for the product line context. It would not be possible to generate products with just one of the conditions, unless some refactoring is done previously.

The next piece box scenario shows that Edicts implementation might hinder the maintenance effort adding problems like code scattering and duplication. Additionally, we found other problems, when considering code tangling and size as well. We discuss all issues more specifically throughout this paper and we also propose alternative solutions and evaluate them all.

III. Idioms

In this section we present two new idioms addressing the code repetition problem of Edicts, dicussed in Sec. II: one for AspectJ, based on aspect inheritance, and another in CaesarJ [8], based on static and dynamic aspect deployment. A key concept used throughout this paper called driver. It is a mechanism (or a set of mechanisms) responsible for providing information about whether a feature should be enabled or not at runtime. These mechanisms vary from simple GUIs (with user inputs) to complex ones like sensors that decide by themselves.

A. Layered Aspect

We defined another idiom, called Layered Aspect, addressing the code duplication problems found in Edicts discussed in Section V. The new idiom relies on the *adviceexecution* pointcut for intercepting the feature code and controling its activation. First, we implement the feature as if it is bound statically. Dynamic binding of the feature is enabled by another aspect that intercepts the pieces of advice of the feature and skips their execution if the feature is turned off. If that aspect is included to the build, the feature is bound dynamically, otherwise it is bound statically.

For example, the next-piece-box feature can be implemented by a single aspect that combines the functionality of the aspects *EdictsNextPieceStatic* (Listing 3) and *EdictsNextPiece-Abstract* (Listing 1). Implementation of larger features can be split into multiple aspects located in the same package. Dynamic binding of the next-piece-box feature is implemented by the aspect *AdviceExecutionNextPieceMethod* (Listing 4). The pointcut *driver()* (Line 2) defines the condition when the feature should be actived. When the driver condition evaluates to false, the feature must be deactivated. This is achieved by the advice in Lines 4-9, which intercepts the pieces of advice of all aspects implementing the feature under the condition *!driver()* and skips their execution by not calling *proceed()*. The pieces of advice of the feature are selected by means of the *adviceexecution* pointcut filtered by the package where the aspects of the feature are located. The *driver()* pointcut is left abstract in *AdviceExecutionNextPieceMethod* and is defined in a concrete driver aspect *AdviceExecution-DriverNextPieceMethod* (Listing 5).

This solution avoids duplication of feature code and localizes the driver code in just two aspects. Adding new drivers is not problematic either. We just need to create another aspect like the one of Listing 5 and implement the desired new driver.

Listing 4. Controlling activation of a feature.

```
1: public abstract aspect AdviceExecutionNextPieceMethod {
2:   abstract pointcut driver();
3:
4:   void around(): adviceexecution()
5:    && within(tetris.ui.adviceexecution.*)
6:    && !within(tetris.ui.adviceexecution
7:    .AdviceExecutionNextPieceMethod) && !driver() {
8:       // skip by not calling proceed
9:   }
10:}
```

Listing 5. Concrete driver code.

```
1: public aspect AdviceExecutionDriverNextPieceMethod extends
2:   AdviceExecutionNextPieceMethod {
3:   ...
4:   pointcut driver() : if (AdviceExecutionNextPieceMethod
5:    .userChoice.equals("Yes"));
6: }
```

Notice that the advice defined in Lines 4-9 works well only for before and after advice. Turning off the around advice is more complicated, because when we turn off the around advice of a feature, we must restore the base code overriden by it. Since AspectJ does not provide a possibility to access the *proceed* joinpoint 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 must be turned off one by one.

One may suggest this solution is problematic for turning off intertype declarations of a feature, since we apply the

[3]If the bodies of the advice are large we can avoid their duplication by moving them into shared methods, declared in the abstract aspect, but then we still need to duplicate advice declarations and method calls.

driver only on advices. Note that the class members introduced by the intertype declarations can be accessed only from the feature code. Therefore, if the feature is disabled, there are no references to the introduced members.

B. Flexible Deployment

CaesarJ is an aspect-oriented language, which extends AspectJ with support for flexible instantiation and deployment of aspects [8]. Instead of introducing special construct for aspects, CaesarJ allows to declare pointcuts and pieces of advice directly in classes.[4] For example, the functionality of the next-piece-box feature is implemented in the class *NextPieceCaesarJ* (Listing 6).

Flexible aspect deployment can be exploited for implementation of flexible binding time of features. In order to implement dynamic binding time for the next-piece-box feature based on CaesarJ, we create a statically deployed class *NextPieceChoiceCaesarJ* (Listing 7) to represent the driver.[5] The class contains the driver code that instantiates and deploys an instance of *NextPieceCaesarJ*, whenever the next-piece-box feature needs to be activated. In our example, the driver class intercepts execution of the *startApp* method (Line 4) and presents the user with the choice to activate feature (Line 7).

A feature can be bound statically by statically deploying the class containing the feature code. Alternatively we can define a statically deployed subclass, like class *NextPieceStaticCaesarJ* in Listing 8. We can switch between static and dynamic binding of the feature by including either *NextPieceChoiceCaesarJ* or *NextPieceStaticCaesarJ* to the build.

Listing 6. Feature code in CaesarJ.

```
1:  public cclass NextPieceCaesarJ {
2:
3:    private NextPieceBox nextPieceBox;
4:
5:    pointcut nextPiece(TetrisCanvas canvas):
6:     execution(private void TetrisCanvas.setupLayout(..))
7:      && this(canvas);
8:
9:    after(TetrisCanvas canvas): nextPiece(canvas) {
10:    nextPieceBox = new NextPieceBox(...);
11:
12:   ...
13: }
```

Listing 7. Dynamic deployment of a feature in CaesarJ.

```
1:  public deployed cclass NextPieceChoiceCaesarJ {
2:
3:   pointcut startApp(TetrisMidlet midlet):
4:    execution(void TetrisMidlet.startApp()) && this(midlet);
5:
6:   before(TetrisMidlet midlet): startApp(midlet) {
7:    int userChoice = JOptionPane.showConfirmDialog(...)
8:    if (userChoice == JOptionPane.YES_OPTION) {
9:     deploy new NextPieceCaesarJ();
10:   }
11:  }
12: }
```

[4]Classes with CaesarJ-specific features are declared with the keyword *cclass*, because the keyword *class* is reserved for pure Java classes.

[5]Statically deployed classes (declared with keyword *deployed*) are singletons analogous to AspectJ aspects.

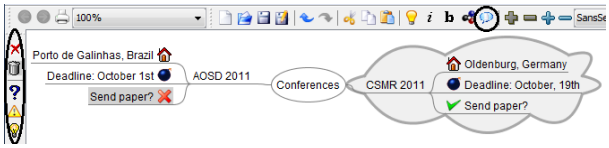


Fig. 2. Mind map constructed in Freemind.

Listing 8. Static deployment of a feature in CaesarJ.

```
1: public deployed cclass NextPieceStaticCaesarJ
2:  extends NextPieceCaesarJ {
3: }
```

Like the Layered Aspect idiom, the solution with CaesarJ does not duplicate feature code and localizes the driver code in a single class.

IV. Study Settings

In this section we detail the study configuration. Firstly, we explain the three case studies[6], and then we discuss the goals and the research questions we intend to investigate in our study. Further, this section outlines the metrics used in the assessment. Finally, we explain the assessment procedures.

A. Case studies

This section presents the case(s) studies in which the features were extracted[7] and implemented with static and dynamic binding times. In order to facilitate the maintenance, we divided the feature extraction throughout some aspects or *cclass*. This way, we can maintain code cohesion and modularity.

1) Freemind: Freemind[8] is an open source system used to construct diagrams to organize ideas. Figure 2 illustrates a mind map in Freemind. It organizes the information of upcoming Software Engineering conferences. As showed in Figure 2 (emphasized by black circle), the map nodes may contain icons and clouds. For example, we used a cloud in the CSMR 2011 deadline to alert us it is coming up, and we also used some icons to increase the node representativeness. Icons and clouds represent the features we extracted in this work.

The features Icons and Clouds are crosscutting and scattered throughout the layers of the architecture. For example, when clicking on the buttons highlighted in Figure 2, the system must add/remove icons and clouds to/from the selected node in the map.

After extracting both features, it is possible to generate products in which the user may enable and disable icons and clouds dynamically. Therefore, our extractions should lead towards all combinations of products with static and dynamic binding times.

[6]We also use Tetris J2ME as a case studies, but we assume its only feature is already explained in Section II

[7]Except to BerkeleyDB, since the features were already extracted to aspects [10]

[8]http://freemind.sourceforge.net/

2) ArgoUML: ArgoUML is an open source UML modeling tool that includes support for all standard UML 1.4 diagrams[9]. The application code is separated into subsystems that have different responsibilities. The subsystems are organized in layers and can have a Facade class which is used by all other subsystems.

For this system, we focused firstly on the Notation subsystem. This subsystem is responsible for defining the notation language used in the UML diagrams. Two notations are provided: UML 1.4 and Java. The feature type of UML 1.4 and Java is OR, because both may be present in the final product. Therefore, three products may be generated: (i) UML 1.4-only (static); (ii) Java-only (static); and (iii) both UML 1.4 and Java (dynamic).

We also considered the Guillemets optional feature. It is responsible for showing the symbols "<<" ">>" to accommodate the stereotypes of classes in the diagrams. Although Guillemets seems to be a simple feature, it is actually scattered throughout many modules of ArgoUML. Thus, it is hard to extract the Guillements feature code to aspects.

3) BerkeleyDB: BerkeleyDB[10] is an open source database that stores arbitrary key/data pairs as byte arrays, and supports multiple data items for a single key. We implemented flexible binding time for seven features that comprehend about database functionalities.

The first feature, named EnvironmentLock, includes the functionality responsible for locking writing operations on the database. The Checksum feature provides a mechanism to detect any corruptions in IO path. There is an interaction in some features like TruncateDB responsible for dropping all tables, views, indexes and sequences, and DeleteOP responsible for deleting the whole database. The fifth feature is about caching, named LookAhead, it caches bodies of information before they are requested by the user, that is useful to improve information retrieval. Flexible binding time was introduced in two more features: Evictor and NIO.

B. Goal

The main goal of our study consists of assessing idioms to implement features with flexible binding times. Notice that such assessment is based on modularity, so we are concerned about aspects like (i) code duplication, (ii) features and drivers code scattering, (iii) tangling between the driver code and the feature code and (iv) the implementation size.

C. Questions

In what follows, we detail the questions that are investigated in this paper.

Which idiom contributes to reduce:

1) *the code duplication when implementing binding time flexibility?*
2) *the driver and feature code scaterring?*
3) *the tangling between the driver and feature code?*
4) *the lines of code and number of components?*

[9]http://argouml.tigris.org/

[10]http://www.oracle.com/technology/products/berkeley-db/index.html

D. Metrics

In order to answer the questions just presented, we evaluate different implementations of the case studies through a metrics suite (Table I) that is widely adopted in different case studies [11], [12], [13].

To achieve such evaluation, we use Pairs of Clone Code (PCC) in Section V-A to answer Question 1, as it may indicate a design that could harm maintenance [14], [15]. In contemplation of answering Question 2, we use DOSC [11] and CDC [13] in Section V-B to measure the feature implementation scattering. We use both of them because we realized that some idioms have similar DOSC of the feature implementation, even though they differ significantly with respect to the number of components required to implement the same feature. Therefore, metrics DOSC and CDC complement each other. In order to respond Question 3, we measure the tangling between driver and feature code considering the DOTC [11] metric in Section V-C. Finally, Source Lines of Code (SLOC) and Vocabulary Size (VS) are well known metrics for quantifying a module size and complexity. In Section V-D we answer Question 4 measuring the size of each idiom in terms of lines of code and number of components. For our context, lower values for the given metrics implies better results, for all these metrics.

E. Assessment procedures

Now, we describe how we perform our study. It is divided into three phases: (i) selection of representative applications and features; (ii) extraction of the features to two AspectJ-based idioms and one CaesarJ-based; and (iii) qualitative and quantitative assessments of each idiom.

Selection of applications and features. In order to generalize the results of our study to other contexts, our applications and features must be representative. We believe the features considered are representative for the following reasons: they have different sizes, architectures, types, granularity, and complexities. For example, we analyzed two[11] types of feature: optional and OR. Notation feature is coarse-grained whereas Guillemets is fine-grained. The next piece box feature is small and its complexity is low. In their turn, Icons and Clouds are much bigger and crosscut the entire Freemind architecture, which increases significantly their complexity. Besides, the features we consider in this paper exercise many AspectJ constructs, such as advice (before, after, and around), inter-type declarations of methods, attributes, and changes to the hierarchy of classes, and also aspect inheritance. Additionally, three case studies consist of real applications. Last but not least, it is important to note that the features had to make sense for the binding time context so that they were selected accordingly.

Extraction. After the selection phase, we extracted the features by using three idioms for binding time flexibility:

[11]The alternative type was not considered because it makes no sense for the dynamic binding time. According to Feature Modeling [16], if two features are alternatives, both cannot exist in the same product. Therefore, there is no way to choose between them dynamically.

TABLE I
METRICS SUITE

Attributes	Metrics	Definitions
Separation of Concerns	Degree of Scattering over Components (DOSC)	A measure of the variance of the concentration of a concern over the features.
	Concern Diffusion over Components (CDC)	Number of components that contribute to the implementation of a concern.
	Degree of Tangling within components (DOTC)	A measure of the variance of how dedicated the components are to the features of the program.
Size	Source Lines of Code (SLOC)	Number of source lines of a component (e.g., classes or aspects).
	Vocabulary Size (VS)	Number of components (classes, interfaces, and aspects) of the system.
Clone	Pairs of Clone Code (PCC)	Number of pairs of clone code resulting from the modularization of drivers and features using a particular idiom.

Edicts (Section II), Layered Aspect (Section III-A), and Flexible Deployment (Section III-B). All possible combinations of products and binding times were generated for each idiom and exploratory tests were performed in order to guarantee that our extractions did not cause problems to the applications.

Assessment. We performed our study assessment not only quantitatively by using the metrics previously described, but also qualitatively. Therefore, we discuss the advantages and disadvantages of each idiom by analyzing (i) the results of the metrics as well as (ii) the design quality provided by each idiom regarding scenarios related to modularity.

V. EVALUATION

In this section we evaluate the idioms just presented. The evaluation is aimed at answering the questions outlined in Section IV-C, so that each question is answered in the discussion of the subsequent subsections.

A. Cloning

This section aims at answering Question 1 from Section IV-C. Table II summarizes the results of the Pairs of Clone Code (PCC) metric. The minimum clone length (in tokens) used was 40, which means that to be considered cloned, two pairs of code must have at least 40 similar tokens. Pairs of similar code that have 39 tokens are not considered cloned, so that they are discarded by the PCC metric. Results using less than 40 similar tokens are discarded due to the frequently appearance of undesired clones, for instance, package path, Java imports or component name. Tools like CCFinder [12] uses 50 as a default number for the minimum clone length. We used 40 due to some uninteresting results we found when using 50: for many features and idioms, the pairs of clones were zero. In other words, if we use a higher number of similar tokens, we miss some interesting clones. On the other hand, if we use a lower number of similar tokens, we find many interesting clones.

As expected, since the use of Edicts implies feature code duplication, the number of pairs of clones is high for this idiom, especially for fine-grained features like Icons. In this case, many advices are duplicated in both dynamic and static aspects. There is no need to duplicate the pointcut, because it is inherited from the shared abstract aspect.

[12]http://www.ccfinder.net/

TABLE II
PAIRS OF CLONES FOUND IN EACH IDIOM.

	Edicts	Layered Aspect	Flex Deployment
Icons	22	1	1
Clouds	14	2	2
Notation	8	2	8
Guillemets	5	6	0
Next Piece Box	0	0	0
Environment Lock	3	0	0
Checksum	9	0	0
TruncateDB	11	2	2
DeleteOp	9	1	1
LookAhead	2	0	0
Evictor	5	0	0
NIO	0	0	0

Layered Aspect has low PCC rates because it does not duplicate driver and feature code, because we do not need to clone the aspect for static and dynamic contexts.

Flexible Deployment has low PCC rates too. This solution does not duplicate the driver, which is located in just one class. Like Layered Aspect, Flexible Deployment does not need to duplicate feature code, because it is not necessary to clone the class containing the feature code for static and dynamic contexts.

Some cloning is still detected for Layered Aspect and Flexible Deployment due to things like very long package paths specified in pointcuts or Java imports used in more than one component. Although we use 40 similar tokens as a parameter to the CCFinder tool, we still do not completely exclude such cases.

B. Scattering

As mentioned in Section IV-D, we used CDC and $DOSC$ to analyze scattering of the feature and driver code for each idiom. By using these metrics, we intend to answer Question 2 from Section IV-C. In particular, we study the driver scattering in Section V-B1 and the feature scattering in Section V-B2.

1) Driver: In this section, we discuss the driver code scattering. Table III presents the $DOSC$ of the driver code.

TABLE III
DRIVER DOSC

	Icons	Clouds	Notation	Guillemet	NextPiece	EnvLock	Checksum	TruncateDB	DeleteOp	LookAhead	Evictor	NIO
Edicts	0,910	0,829	0,000	0,819	0,051	0,002	0,002	0,002	0,453	0,002	0,002	0,002
Lay. Aspect	0,001	0,703	0,000	0,485	0,254	0,503	0,501	0,002	0,503	0,002	0,002	0,002
Flex. Deploy	0,001	0,001	0,000	0,000	0,051	0,002	0,002	0,002	0,002	0,002	0,002	0,000

Edicts has the highest rates due to the *if* statements scattered throughout the aspects. The results are especially bad for large features, implemented by multiple aspects. Such design is error-prone, because forgetting an *if* statement like that may cause runtime crashes to the system.

Notice that the scattering of the driver code is lower for the Layered Aspect idiom. However, when considering fine-grained features with *around* advice, the driver code $DOSC$ is higher. As explained in Section III-A, such pieces of advice must be handled individually, which leads to driver scattering through the feature code.

Flexible Deployment has the best results for $DOSC$. This idiom does not scatter the driver code, because we have only one class responsible for the implementation of the driver. Therefore, there is no need to scatter the driver code throughout the feature code.

2) Feature: Now, we discuss the feature scattering. As can be seen in Table I, the $DOSC$ metric depends of the number of components used to implement the feature. Therefore, we consider two perspectives. From a package perspective, the feature is well modularized as its implementation is in a single package. Therefore, there is no feature scattering at the package level. On the other hand, one idiom may use multiple aspects or classes to implement a feature, leading to a higher feature scattering considering the number of components of its implementation. According to the results showed in Table IV, some features are scattered at the component level, which means that their code seems to be equally distributed throughout the aspects or classes responsible for implementing them. In other words, there is no particular aspect or class that concentrates the majority of the feature code.

Notice that such scattering occurred in all idioms, which is insufficient to help us answering the question about which idiom reduces the feature scattering. This way, we applied another metric to measure the scattering through a different perspective. The CDC allowed us to identify potential differences among the idioms (Table V).

The Edicts idiom has the worst results for most of the features. That happens because we may have three aspects for each aspect used to extract the feature code. One for static context, other for dynamic context and the last one to group the pointcuts used by the other two, as explained in Section II.

Layered Aspect has only one aspect, for each aspect used to extract the feature, which leads to low feature code scattering relatively to the other two idioms. This way, Layered Aspect has the best results for CDC metric.

Flexible Deployment may have two *cclass* for each aspect used to extract the feature, as explained in Section III-B. Therefore, the CDC metric shows that this idiom scatters feature code more than Layered Aspect but less than Edicts.

Despite the $DOSC$ similarities for all idioms, taking the Icons feature into consideration, we have $CDC(Edicts) = 23$; $CDC(LayeredAspect) = 11$; $CDC(FlexibleDeployment) = 10$. Now, we can see that maintaining the Icons feature is more difficult for the Edicts idiom, since there are more components to consider.

C. Tangling

This section answers Question 3 by investigating the extent of tangling between a feature and its driver. According to the principle of separation of concerns, one should be able to implement and reason about each concern independently. In this work, we assume that *the greater is the tangling between the feature and its driver, the worse is the separation of those concerns*. As mentioned in Section IV-D, we measured the Degree of Tangling within Components ($DOTC$) [11]. Table VI shows the average $DOTC$ regarding the interactions between features and drivers.

Edicts has the highest $DOTC$ rates in the majority of the cases, because the driver, implemented by *if* statements, is tangled with the pieces of advice containing the feature code.

Layered Aspect tangles driver and feature code for some features. It occurs because we still have to introduce driver code into the *around* advice of the feature code.

lexible Deployment leads to the lowest tangling for the most of the features. This occurs because this idiom modularizes the driver code within a single class. Unlike the other two idioms, Flexible Deployment keeps the driver code independent of the feature code.

D. Size

The fourth question 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. Table VII presents the results of the $SLOC$ metric.

As explained in Section II, in order to support flexible binding time of a feature, the Edicts idom introduces two additional aspects 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.

The Layered Aspect and Flexible Deployment achieve better results than Edicts due to the absence of feature code duplication. The driver is not duplicated either.

Now, we analyze the size from the number of components perspective by using the VS metric. The results are detailed in Table VIII. In general, we observed that when few aspects are necessary to implement the feature, all idioms have similar number of components. However, the difference between Layered Aspect and the other two idioms becomes visible the larger is the number of aspects used to implement the features. Edicts and Flexible Deployment produce more components,

TABLE IV
FEATURE DOSC

	Icons	Clouds	Notation	Guillemet	NextPiece	EnvLock	Checksum	TruncateDB	DeleteOp	LookAhead	Evictor	NIO
Edicts	0,956	0,945	0,719	0,930	0,747	0,570	0,608	0,503	0,749	0,628	0,573	0,664
Lay. Aspect	0,877	0,880	0,879	0,833	0,318	0,200	0,390	0,002	0,488	0,002	0,218	0,478
Flex. Deploy	0,878	0,887	0,612	0,839	0,227	0,050	0,026	0,038	0,029	0,067	0,171	0,584

TABLE V
CDC

	Icons	Clouds	Notation	Guillemet	NextPiece	EnvLock	Checksum	TruncateDB	DeleteOp	LookAhead	Evictor	NIO
Edicts	23	22	5	18	3	3	3	2	6	3	3	3
Lay. Aspect	11	11	4	8	3	3	3	2	4	2	2	2
Flex. Deploy	10	19	5	18	3	3	3	2	3	3	3	3

TABLE VI
DOTC

	Icons	Clouds	Notation	Guillemet	NextPiece	EnvLock	Checksum	TruncateDB	DeleteOp	LookAhead	Evictor	NIO
Edicts	0,225	0,177	0,182	0,215	0,291	0,137	0,238	0,205	0,150	0,167	0,213	0,260
Lay. Aspect	0	0,064	0,249	0,090	0,124	0,310	0,151	0	0,081	0	0,420	0,260
Flex. Deploy	0	0	0	0	0	0	0	0	0	0	0	0

TABLE VII
SLOC

	Icons	Clouds	Notation	Guillemet	NextPiece	EnvLock	Checksum	TruncateDB	DeleteOp	LookAhead	Evictor	NIO
Edicts	836	616	180	422	95	225	445	304	626	149	242	34
Lay. Aspect	459	324	69	222	91	126	215	158	309	79	118	20
Flex. Deploy	435	384	160	264	48	135	261	175	229	102	136	30

because both need additional aspects or classes to implement dynamic and static binding time.

E. Discussion

In this section, we discuss the advantages and disadvantages of all three idioms.

Listing 9. Trying to turn off around advice with *adviceexecution*.

```
1:  Object around(): adviceexecution()
2:   && within(tetris.ui.adviceexecution.*)
3:   && !within(tetris.ui.adviceexecution
4:    .AdviceExecutionNextPieceMethod) {
5:    if (driver.equals("true")) {
6:     return proceed();
7:    }
8:     return null;
9:   }
10: }
```

In summary, Edicts gives the worst results considering the metrics. This idiom causes modularity problems in the most of the cases, because it clones advices, and the driver code is scattered throughout the aspects and tangled with feature code. In addition, Edicts introduces more components than the other idioms due to its need to have static, dynamic, and abstract aspects. These problems are harmful to software maintenance. If we forget to introduce the driver inside a piece of advice, we could have a runtime exception in case the feature is disabled. Maintaining the feature code is more time consuming and error prone, since we have to fix the same problem twice for a static and a dynamic aspect.

In contrast, Layered Aspect and Flexible Deployment give good results considering the metrics. The advice and pointcuts are not duplicated, the driver code is neither scattered nor tangled throughout the aspects or classes like in case of Edicts.

Layered Aspect causes driver scattering when the feature code contains *around* advice. This way, we cannot use only the AspectJ *adviceexecution* pointcut, otherwise we would skip not only the feature code but also the base code in case the feature is disabled dynamically, as showed in Listing 9. The aspect inheritance solution (Section III-A) introduce driver code to those kind of advice, which can lead to a problem if we want to remove the driver from many advices, for example.

Despite of the good metric results, Flexible Deployment has some deficiencies that are not shown by the metrics. First, *priviledged cclass*es are not supported. Therefore, we have to create getters and setters for private attributes, and change the visibility of some methods to public. This can lead to undesired exposure of internal details, which can hinder object-oriented encapsulation. Second, CaesarJ uses wrapper classes instead of intertype declarations to add the new elements of a feature. Although wrappers avoid various modularity problems of intertype declarations [8], they introduce some syntactic overhead in the feature code.

VI. THREATS

In what follows, we discuss some threats that might appear in our study.

The first threat regards to the feature extraction. All the features were not extracted by the same person, BerkeleyDB was already extracted with aspects and four people extracted the other features. This way, we may have introduced a bias which could alter the evaluation. However, we reviewed all the implementations in order to assure that a bias is not introduced.

The second threat concerns the performance of the idioms. We do not evaluate performance throughout this paper, even though this is an important point, mainly for the dynamic context. We intend to do this evaluation as a future work.

VII. 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.

Features with Flexible Binding Times. An alternative proposal considers *conditional compilation* as a technique to

TABLE VIII
VS

	Icons	Clouds	Notation	Guillemet	NextPiece	EnvLock	Checksum	TruncateDB	DeleteOp	LookAhead	Evictor	NIO
Edicts	597	596	1639	1645	20	405	405	405	408	405	405	405
Lay. Aspect	585	585	1640	1636	20	405	405	404	406	404	404	404
Flex. Deploy	584	593	1639	1641	20	405	405	405	405	405	405	405

implement features with flexible binding times [17]. Such a work discusses how to apply *conditional compilation* in real systems like operating systems. Likewise we described in our work, developers need to decide what features must be included to compose the product and their respective binding times. Listing 10 illustrates an example using conditional compilation with the next piece box example discussed in Section II. However, the work concludes that, in fact, conditional compilation is not a very elegant solution. Hence, when we have complex variation points, the situation becomes even worse.

Listing 10. Binding time with conditional compilation.

```
//#if NEXT_PIECE_BOX
  //#if STATIC
  //#  nextPieceBox = new NextPieceBox(...);
  //#else
  //#  if (getUserChoice()) {
  //#    nextPieceBox = new NextPieceBox(...);
  //#  }
  //#endif
//#endif
```

Tanter [18] complements the work with CaesarJ in which he provides a precise and expressive scoping of aspects at deployment time. This mechanism is called *deployment strategies*. This work also claims that the actual approach to restrict join points, which an aspect should act, is achieved not elegantly since conditions to the pointcut definitions should be introduced, sacrificing the potential reuse of aspects, as we showed in this paper. Filtering some join points dynamically could be useful because one may want to disable part of a feature (a subfeature, for example), which means that only a part of the aspect should execute.

History Based Aspects. Bodden et al. describe a novel AspectJ language extension, know as *dependent advice*, responsible for aiding optimization of history based aspects [19]. Dependent advice extends the notion of advice with dependency annotations. Thus, a dependent advice is executed if and only if its dependencies are satisfied. Their approach include a whole program analysis that removes advice dispatch code. The authors argue that the results they obtain significantly reduce the runtime overhead of history-based aspects. Such a work can be adopted to provide the driver implementation of our work, so that by using dependent advice, one can precisely specify what advice must be executed when the driver condition (dynamic binding time) is satisfied. Therefore, this work proposes a syntax extension for AspectJ to restrict aspects execution in a dynamic way. Hence, we intend to use dependent advice as future work in order to implement features with flexible binding times.

Empirical Case Studies. There are some empirical case studies about extracting features to aspects [20], [21], [22]. For instance, Kastner et al. [10] conducted a case study that restructured the database system Berkeley DB into 38 features. As stated previously, we implemented flexible binding time into seven extracted features from this work. They used AspectJ to implement the features. However, the authors used only basic AspectJ constructs, such as static introductions and method extensions, whereas besides that, we used other structures, like aspect inheritance and *adviceexecution*. They concluded that AspectJ is not suitable to implement features since it decreases the code readability and maintainability. It is important to note that they claim this AspectJ unsuitability is related to refactored features from legacy applications. In contrast, we showed that idioms like Layered Aspect may be elegantly applied and worthwhile for implementing features with flexible binding times.

Aspect Inheritance. Another proposal to implement flexible binding time into features considers aspect inheritance [23]. 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 all feature code and an abstract pointcut definition (Listing 11), then we associate this driver like a condition with the advice, as illustrated in Lines 5, 8 and 11. Furthermore, we create two aspects that inherit from the abstract one, in order to implement the concrete driver (Listing 12). Differently from Edicts, Aspect Inheritance avoid feature code duplication. However, this solution is not suitable, because the driver scattering problem arises in those aspects.

Listing 11. Abstract driver and the next piece box feature.

```
1:  public abstract aspect NextPieceBox {
2:   abstract pointcut driver();
3:
4:   pointcut nextPiece(TetrisCanvas canvas):
5:      ... && driver();
6:
7:   pointcut infoBoxes(Graphics g,
8:    TetrisCanvas canvas): ... && driver();
9:
10:  pointcut paintOnce(Graphics g,
11:   TetrisCanvas canvas): ... && driver();
12:    ...
13: }
```

Listing 12. Extending NextPieceBox to define the concrete driver pointcut.

```
1:  public privileged aspect DynamicNextPieceBox
2:   extends NextPieceBox {
3:
4:   pointcut driver(): if (userChoice.equals("Yes"));
5:  }
6:
7:  public privileged aspect StaticNextPieceBox
8:   extends NextPiece {
9:
10:  pointcut driver() : if (true);
11: }
```

VIII. Concluding remarks

This paper presented a study about implementing binding time flexibility into features. The first idiom is Edicts which showed some modularity problems like feature code duplication and driver code scattering and tangling. This idiom can be error-prone and can hinder the maintenance effort in some cases. The second idiom is Layered Aspect which presented better results regarding to modularity, it does not tangle nor scatter driver code like Edicts and it does not duplicate feature code. The third one is called Flexible Deployment, this solution isolates the driver code completely from the feature code, there is no code duplication as well.

In order to evaluate the idioms, we assessed them targeting feature modularity, based on qualitative and quantitative analysis. In addition, we discussed the idioms and presented the problems found.

IX. Acknowledgement

We would like to thank CNPq, a Brazilian research funding agency, and National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, for partially supporting this work. In addition, we thank Alessandro Garcia from PUC-Rio and SPG members for feedback and fruitful discussions about this paper. Finally, we thank Venkat Chakravarthy and Eric Eide for providing some Edicts examples.

References

[1] K. Pohl, G. Bockle, and F. J. van der Linden, *Software Product Line Engineering*. Springer, 2005.

[2] V. Chakravarthy, J. Regehr, and E. Eide, "Edicts: Implementing Features with Flexible Binding Times," in *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*. New York, NY, USA: ACM, 2008, pp. 108–119.

[3] M. Rosenmüller, N. Siegmund, G. Saake, and S. 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)*. New York, NY, USA: ACM, 2008, pp. 3–12.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting Started with AspectJ," *Communications of the ACM*, vol. 44, no. 10, pp. 59–65, October 2001.

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

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

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

[8] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, "An overview of caesarj," *Lecture Notes in Computer Science : Transactions on Aspect-Oriented Software Development I*, pp. 135–173, 2006.

[9] L. Rajlich, "How farmville scales to harvest 75 million players a month," September 2010, http://highscalability.com/blog/2010/2/8/how-farmville-scales-to-harvest-75-million-players-a-month.html.

[10] C. Kastner, S. Apel, and D. Batory, "A Case Study Implementing Features Using Aspectj," in *Proceedings of the 11th International Software Product Line Conference (SPLC'07)*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 223–232.

[11] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do crosscutting concerns cause defects?" *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, 2008.

[12] R. Bonifácio and P. Borba, "Modeling scenario variability as crosscutting mechanisms," in *Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD'09)*. ACM, 2009, pp. 125–136.

[13] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa, "Modularizing Design Patterns with Aspects: A Quantitative Study," in *LNCS Transactions on Aspect-Oriented Software Development I*. Springer, 2006, pp. 36–74.

[14] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering (ASE)*, vol. 3, no. 1, pp. 77–108, 1996.

[15] I. 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)*, 1998, pp. 368–377.

[16] K.-C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA). Feasibility Study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, November 1990.

[17] E. Utrecht, G. Florijn, and E. Dolstra, "Timeline variability: The variability of binding time of variation points," in *Proceedings of the Workshop on Software Variability Management (SVM'03)*, 2003, pp. 119–122.

[18] Éric Tanter, "Expressive scoping of dynamically-deployed aspects," in *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*. New York, NY, USA: ACM, 2008, pp. 168–179.

[19] E. Bodden, F. Chen, and G. Rosu, "Dependent advice: a general approach to optimizing history-based aspects," in *Proceedings of the 8th ACM international conference on Aspect-oriented software development (AOSD'09)*. New York, NY, USA: ACM, 2009, pp. 3–14.

[20] C. Zhang and H.-A. Jacobsen, "Quantifying aspects in middleware platforms," in *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD'03)*. New York, NY, USA: ACM, 2003, pp. 130–139.

[21] Y. Coady and G. Kiczales, "Back to the future: a retroactive study of aspect evolution in operating system code," in *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD'03)*. New York, NY, USA: ACM, 2003, pp. 50–59.

[22] F. Hunleth and R. K. Cytron, "Footprint and feature management using aspect-oriented programming techniques," *SIGPLAN Not.*, vol. 37, no. 7, pp. 38–45, 2002.

[23] M. Ribeiro, R. Cardoso, P. Borba, R. Bonifácio, and H. 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)*, Fortaleza, Cear'a, Brazil, 2009, pp. 1–6.

A. Online Appendix

We invite researchers to replicate our study. All results are available at: http://twiki.cin.ufpe.br/twiki/bin/view/SPG/FlexibleBindingTime.