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 Universitt Darmstadt
Darmstadt, Germany
gasiunas@st.informatik.tu-darmstadt.de

Lucas Satabin
Technische Universitt 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;**

I. Introduction

In order to obtain significant improvements in time to market, maintenance cost, productivity and quality of products, companies are adopting the Software Product Line (SPL) development paradigm. SPL encompass a family of software-intensive systems developed from reusable assets. By reusing such assets, it is possible to construct a large scale of products through specific features defined according to customers requirements [1].

Due to specific client requirements, it is important to define whether some features are included or not. In this context, the binding time of a feature is the time at which one decides to include or exclude 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 desktop systems, 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 aiming at supporting the implementation of features with flexible binding times in a modular and convenient way [2]. In this work, we used Edicts to implement flexible binding time into several features from four systems (Tetris J2ME[1], Freemind [5], ArgoUML [6] and BerkeleyDB [7]). Although the authors claim that Edicts are modular, no assessment was performed with respect to modularity. Actually, the situation gets even worse: we observed problems when using the Edicts idiom, such as code duplication. This way, the task of maintaining the system becomes time consuming and error-prone.

With the aim at evaluating and comparing Edicts with other idioms, we also implemented flexible binding time into the same features with another AspectJ-based idiom developed by us, that relies on adviceexecution pointcut and aspect inheritance provided by AspectJ. The third idiom is based on CaesarJ [8], which allows flexible deployment.

In Sections III-C and III-D 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 based on several case studies considering the three idioms with regard to cloning, scattering, tangling and size of the resulting implementation. Furthermore, we show that some problems, which affect directly the maintenance effort, were encountered in all of the idioms. Finally, in Section V-E, we discuss the idioms qualitatively.

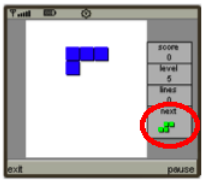
II. Motivating Example

It is important to define a product line in which a feature has different binding times [9] for a number of reasons. For instance, the Facebook Farmville game needs to ”dynamically turn off calls back to the platform” at peak times ”since performance can be variable” [10]. On the other hand, when considering enterprise social network environment, this call back feature might not need to be turned off dynamically. So

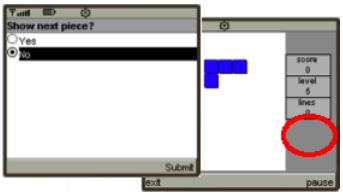
[1]http://kiang.org/jordan/software/tetrismidlet/

we suggest static bind time for the latter and dynamic binding time for the first.

To implement flexible binding time, we could use the edicts idiom which is supposed to avert modularity problems like code scattering and duplication. To illustrate how this works, consider the next piece box feature from Tetris J2ME. This scalable feature is responsible for showing the next piece box that the game is about to drop on the screen. Figures 1(a) and 1(b) show both dynamic and static binding time scenarios.



(a) Static next piece box.



(b) Dynamic next piece box.

Fig. 1. Two products of the Tetris game: specific binding times for the feature.

Following the edicts idiom[2], we must create one aspect for dynamic next piece box, another for static next piece box and one abstract aspect which contains the pointcut used for static and dynamic context. Listing 1 shows part of the *EdictsNextPieceDynamic* aspect and Listing 2 shows the *EdictsNextPieceAbstract* aspect. The pointcut *nextPiece* inherited from *EdictsNextPieceAbstract* intercepts the *setupLayout* method execution from *TetrisCanvas* class to allow the advice defined in Line 6 to initialize the *NextPieceBox* object after the method execution intercepted (Line 8), if the user choice is equal to "Yes". Analogously, the advice defined in Line 16 paints the next piece box after the method execution *paintInfoBoxes* which is intercepted by the pointcut *infoBoxes* inherited from *EdictsNextPieceAbstract*. 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 (Lines 7 and 15), since there is no user decision in such context. 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 this case, we use dynamic binding time, so it is required to include the *EdictsNextPieceDynamic* and *EdictsNextPieceAbstract*. We distinguish both static and dynamic contexts because of the performance overhead introduced by the latter [3].

[2]Edicts work does not claim that patterns are sufficient or necessary to implement all variation points in product lines. In this way, they claim that it is not mandatory to use edicts with patterns.

Listing 1. EdictsNextPieceDynamic aspect

```
1: public privileged aspect EdictsNextPieceDynamic
2:  extends EdictsNextPieceAbstract {
3:
4: private NextPieceBox nextPieceBox;
5:
6: after(TetrisCanvas canvas): nextPiece(canvas) {
7:  if (userChoice.equals("Yes")) {
8:   nextPieceBox = new NextPieceBox(...);
10: }
11:}
12:
13: after(Graphics g, TetrisCanvas canvas):
14:  infoBoxes(g, canvas) {
15:  if (userChoice.equals("Yes")) {
16:   if (nextPieceBox.setPieceType(
17:    canvas.game.getNextPieceType())) {
18:    nextPieceBox.paint(g);
19:   }
20:  }
21: }
22:     ...
23:}
```

Listing 2. EdictsNextPieceAbstract aspect

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

The code duplication between static and dynamic aspects (the advices are repeated) results in maintenance problems since it becomes time consuming and error-prone, specially for larger features. For the Tetris J2ME game, the small feature next piece box has twenty nine lines of feature code in the *EdictsNextPieceDynamic* aspect, twenty three lines may be repeated in the *EdictsNextPieceStatic* aspect resulting in almost eighty percent of feature code clone. The problem gets even worse in the dynamic context. For instance, suppose that a player is about to lose a game. We must add a new kind of condition to decide whether the feature will be executed or not. Dynamically, the system could identify this situation and activate the next piece box feature aiming to help the player. This condition has a completely different implementation when compared to the screen showed in Figure 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 and Driver

In this section, we present the idioms to implement features with flexible binding times[3]. We use the next piece box feature from Tetris J2ME game throughout this section to introduce the idioms. We chose the Tetris case study due to the simplicity and for didactic reasons.

A. Driver

A key concept used throughout this paper is 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.

B. Edicts

Edicts is a technique to implement binding time flexibility of features in a modular and convenient way [2]. Edicts make possible to choose between static and dynamic binding times by using design patterns and aspects. Patterns encapsulate the variation points whereas edicts (aspects) set the binding times of the features.

As showed in Section II, one aspect is responsible for the static binding time, whereas the other one implements the dynamic binding time. The pointcuts used by both dynamic and static aspects are implemented in the abstract aspect. The dynamic aspect also implements the driver, which is the screen from Figure 1(b) and the user input: depending on the user choice, the next piece box is displayed or not.

Notice that we did not create Edicts. Actually, Edicts motivated us to create the other idioms because we found some modularity problems in it.

C. Layered Aspect

We defined another idiom aiming at solving the modularity problems found in Edicts, as showed in Section V. This new idiom relies on the *adviceexecution* pointcut (Listing 3) and aspect inheritance (Listing 4). The *adviceexecution* pointcut intercepts advice executions specified in aspects inside the *tetris.ui.adviceexecution* package, but not inside the *AdviceExecutionNextPieceMethod* since, this aspect has only the driver implementation and around advices that return types. Notice that the advice defined in Line 3 (Listing 3) only intercepts advices *before, after* and *void around*. In order to intercept *around* advices that have a return, we define an abstract pointcut (Line 10). Thus, we associate this pointcut to the corresponding advice and implement it in the *AdviceExecutionDriverNextPieceMethod* aspect (Listing 4). We encourage the usage of aspect inheritance to implement the driver in such situations to avoid runtime exceptions in case the feature is disabled dynamically, since we would have to return *null* instead of returning the *proceed()* (Line 7, Listing 3).

[3]There is an idiom based in aspect inheritance [11] which is explained in Section VI, however we do not consider it for evaluation due to its problems similarity to Edicts

One may suggest this solution is problematic for intertypes, since we only apply the driver for advices. Although, we argue that the methods extracted must be called from inside the feature implementation, we cannot have references to these methods in the rest of the system, otherwise the feature is not extracted correctly. Therefore, if the feature is disabled, there are no references to its intertype methods.

This solution seems to be suitable because there is no code duplication and the driver is not scattered: the task of verifying the driver is localized in just two aspects, which matches all advice of aspects within a particular package. Adding new drivers is not problematic as well. We just need to create another aspect like the one of Listing 4 and implement the desired new driver.

Listing 3. AdviceExecution aspect.

```
1:  public abstract aspect AdviceExecutionNextPieceMethod {
2:
3:  void around(): adviceexecution()
4:   && within(tetris.ui.adviceexecution.*)
5:   && !within(tetris.ui.adviceexecution
6:    .AdviceExecutionNextPieceMethod) && driver() {
7:    proceed();
8:  }
9:     ...
10:  abstract pointcut driver();
11:     ...
12: pointcut checkNextPiece() :
13:  execution(int TetrisMidlet.getNextPieceType());
14:
15: int around() : checkNextPiece() && driver() {
16:  ...
17: }
18:}
```

Listing 4. Aspect Inheritance driver aspect.

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

D. Flexible Deployment

CaesarJ is an aspect-oriented language with a strong support for reusability [8]. It supports object-oriented modularization and aspect-oriented mechanisms like pointcuts and advice.

In order to implement flexible binding time into the feature next piece box based on CaesarJ, we create a deployed *cclass* (Listing 5) to represent the driver. This structure has a pointcut defined in Line 3 that intercepts the *startApp* method execution to allow the advice (Line 6) to show the screen (Figure 1(b)). Depending on the user choice, it deploys the *NextPieceCaesarJ* which contains the feature code (Listing 6). With regard to implement the static bind, we need to extend *NextPieceCaesarJ cclass* with a deployed *cclass* and remove the driver from the product since it makes no sense for static context. Listing 7 shows this implementation. On the other hand, for dynamic bind, we do not need to have the *NextPieceStaticCaesarJ cclass* in the product.

This solution seems to be similar to Layered Aspect as it does not duplicate feature code and it does not scatter driver code. The driver is implemented in just one isolated *cclas*, therefore there is no need to tangle it with feature code.

Listing 5. Flexible Binding Time with Flexible Deployment.

```
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:     midlet, "Show next piece?", "Next Piece",
9:     JOptionPane.YES_NO_OPTION);
10:   if (userChoice == JOptionPane.YES_OPTION) {
11:    deploy new NextPieceCaesarJ();
12:   }
13:  }
14: }
```

Listing 6. Flexible Deployment cclass.

```
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(canvas.infoPanelX, 163,
11:    canvas.infoPanelWidth, 24 * 3 / 2,
12:    TetrisConstants.COLOR_BLACK,
13:    TetrisConstants.COLOR_LIGHT_GREY,
14:    canvas.font);
15: }
```

Listing 7. Flexible Deployment cclass for static bind.

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

IV. Study Settings

In this section we detail the study configuration. Firstly, we explain the three case studies [4], 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 cases studies in which the features were extracted[5] and implemented with static and dynamic binding times.

1) Freemind: Freemind[6] 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.

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

[5]Except to BerkeleyDB, since the features were already extracted to aspects [12]

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

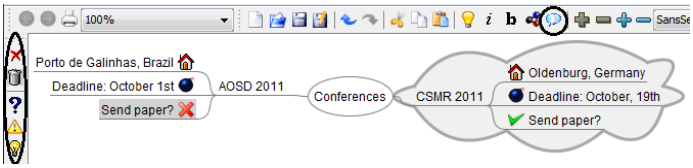


Fig. 2. Mind map constructed in Freemind.

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.

2) ArgoUML: ArgoUML is an open source UML modeling tool that includes support for all standard UML 1.4 diagrams[7]. 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[8] 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

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

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

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?*

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 [13], [14], [15].

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 [16], [17]. In contemplation of answering Question 2, we use DOSC [13] and CDC [15] 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 [13] 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[9] 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: Edicts (Section III-B), Layered Aspect (Section III-C), and Flexible Deployment (Section III-D). 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 follows the questions presented in Section IV-C, so that each question will be answered throughout the discussions of the subsequent sections of the evaluation.

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. Tools like CCFinder [10] 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.

As expected, since the use of Edicts implies feature code duplication, the number of pairs of clones is high for this idiom, specially in fine-grained features like Icons. In this case, many advices are duplicated in both dynamic and static

[9]The alternative type was not considered because it makes no sense for the dynamic binding time. According to Feature Modeling [18], if two features are alternatives, both cannot exist in the same product. Therefore, there is no way to choose between them dynamically.

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

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.

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

aspects. Although, there is no need to duplicate the pointcut, as it is implemented in an abstract aspect which is inherited by the others.

Layered Aspect has low PCC rates because it has no driver and feature duplication, as we do not need to clone the aspect for static and dynamic context.

Flexible Deployment has low PCC rates. This solution does not duplicate the driver, it is located in just one *cclass*. Like Layered Aspect, Flexible Deployment does not need to duplicate feature code, it is not necessary to clone the *cclass* for static context and dynamic context.

B. Scattering

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

1) Driver: In this section, we discuss the driver code scattering. Table III illustrates the feature $DOSC$ we extracted.

Edicts has the highest rates due to the *if* statements scattered throughout the aspects. This idiom is even worse for large features, with many aspects to extract it. Forgetting an *if* statement like that may cause runtime crashes to the system.

Notice that the driver code scattering is lower for the Layered Aspect idiom. Although, when considering fine-grained features that require *around* advices with a return, the driver code $DOSC$ is higher. As explained previously, we need to use aspect inheritance in such cases, which leads to driver scattering through the feature code.

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

2) Feature: Now, we discuss the feature scattering instead of focusing on the driver. According to the results showed in Table IV, some features are scattered, which means that their code seems to be equally distributed throughout the aspects or *cclass* responsible for implementing it. In other words, there is no particular aspect 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-D. 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.

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	0,860	0,000	0,487	0,251	0,423	0,500	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

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	0,721	0,833	0,318	0,288	0,390	0,002	0,577	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

C. Tangling

This section aims to answer Question 3. Therefore, here we investigate how extent is the tangling between a feature and its driver. According to the principle of separation of concerns, one should be able to implement and reason about both concerns independently. In this work, we assume that *the greater is the tangling between the feature and its driver, the lower is the separation of those concerns*. As mentioned in Section IV-D, we used the Degree of Tangling within Components ($DOTC$) [13]. Hence, Table VI shows the average $DOTC$ regarding the interactions between features and drivers.

For the majority of the cases, Edicts has the highest $DOTC$ rates since the driver, implemented by *if* statements, is tangled with advice declarations. This is a problem since it does not separate the concerns and consequently hinder the feature modularity.

Layered Aspect tangles driver and feature code for some features. It occurs because we have to introduce driver code into feature code in case of an *around* advice that has a return appear.

For most of the features, Flexible Deployment leads to the lowest tangling. This occurs because this idiom modularizes the driver code into a single *cclass*. On the contrary of the other two idioms, Flexible Deployment maintain 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 number of components. For this purpose, we use the $SLOC$ and VS metrics. Table VII illustrates the $SLOC$ metric results.

As mentioned on Section III-B, in order to implement flexible binding time, we need three aspects for each aspect used to extract the feature for most of the cases. 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 under 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 need more components because both need aspects or *cclass*, respectively, to implement dynamic and static binding time.

E. Discussion

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

In summary, Edicts had the worst results considering the metrics. This idiom presented modularity problems, for the most of the cases, it clones advices, the driver code is scattered throughout the aspects and tangled with feature code. In addition, Edicts need more components than the other idioms due to its necessity to have static, dynamic and abstract aspects. In this context, those problems are harmful to the system maintenance. If we forget to introduce the driver inside an advice, we could have a runtime exception in case the feature is disabled. Maintaining the feature code could also be problematic and time consuming, since we have to fix twice the same problem for static and dynamic aspect.

In contrast, Layered Aspect and Flexible Deployment had good results considering the metrics. The advices and pointcuts are not duplicated, the driver code is not scattered nor tangled throughout the aspects or *cclass* like Edicts.

Layered Aspect has driver scattering when the feature presents *around* advice that has a return. This way, we cannot use only the AspectJ *Adviceexecution* pointcut, otherwise we would have to return *null* in case the feature is disabled dynamically, as showed in Listing 8. The aspect inheritance solution (Section III-C) introduce driver code to those kind of advice, which can lead to a problem if we want to remove the driver from many advice, for example.

Listing 8. Adviceexecution pointcut.

```
1: Object around(): adviceexecution()
```

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	0,041	0,249	0,055	0,108	0,151	0,121	0	0,060	0	0,345	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

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

```
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: }
```

Despite of the good metric results, Flexible Deployment presents some deficiencies that the metrics do not show. For instance, CaesarJ has no support for Java 5 features (generics, annotations and *assert*) inside a *cclass*, thus we have to adapt the feature code while extracting it. Additionally, *priviledged cclass*es are not supported. Therefore, we have to create getters and setters for private attributes, and change some method visibility to make them public. The internal details become visible to everybody, which can hinder object oriented encapsulation. Moreover, we cannot throw checked exceptions from within a *cclass* advice, we have to introduce a try-catch block inside the advice which has feature code that can possibly throw the exception. This hindrance prevent the system behavior to be as it was before the feature extraction, in case it needs to throw a checked exception.

Finally, CaesarJ has no support for intertype declarations. Thus we have to declare a wrapper *cclass* to add the new elements of a feature and refactor the feature code to use this wrapper whenever one of the new elements are used. This introduces some syntactic overhead in the feature code.

VI. 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 implement features with flexible binding times [19]. 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 9 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 9. Binding time with conditional compilation.

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

Tanter [20] 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 [21]. 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 [22], [23], [24]. For instance, Kastner et al. [12] 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 [11]. 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 10), 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 11). 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 10. 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 11. 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: }
```

VII. 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.

VIII. 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] R. van Ommering, "Building product populations with software components," 2002, pp. 255 – 265.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.