Does AspectJ Provide Modularity when Implementing Features with Flexible Binding Times?

Márcio Ribeiro, Rodrigo Cardoso, Paulo Borba, Rodrigo Bonifácio, Henrique Rebêlo
Informatics Center
Federal University of Pernambuco
Recife, Brazil
{*mmr3, rcaa2, phmb, rba2, hemr*}*@cin.ufpe.br*

***Abstract*—Dynamic configuration has been receiving increased attention in the Software Product Lines community, which means that features with different binding times are becoming important and required. Although aspects have been considered to implement product lines features, some study must be done in order to evaluate whether aspects are also suitable for implementing features with flexible binding times, such as dynamic and static. Thus, this paper presents an exploratory study that investigates whether the most popular AOP language (AspectJ) provides modularity in the flexible binding time context. The results suggest that it depends on the features peculiarities like size and heterogeneity.**

***Keywords*-AspectJ, Binding times, Software Product Lines**

I. Introduction

Dynamic configuration has been receiving increased attention in the Software Product Lines community [1]. Generating products with different binding times is important for several reasons. Products for devices with constrained resources may use static binding time instead of dynamic due to the performance overhead introduced by the latter. Also, a company may want to construct products to deal with client's needs. For example, a thermometer of Brazilian residences should use the Celsius scale. For American residences, the Fahrenheit scale should be used instead. In both cases, the feature scale is selected earlier, e.g., *statically*. There is no way to change it. However, such thermometer should provide both scales for hotels because of the foreign people supposed to use it. In this situation, the hotel guest should be able to change the current scale at any time, e.g., *dynamically*.

Many computing environments provide mechanisms of software and hardware to provide information to enable/disable a feature dynamically. These mechanisms (called "feeders" in this paper) vary from GUIs that ask the user if he/she wants to enable/disable the feature to more complex ones like sensors that decide by themselves.

In this context, maintaining many different features, binding times, and feeders may be a challenge task for the product lines designers. Previous work [2], [3] showed that aspects are useful for implementing product lines features, but they still lack on analyzing if aspects are also suitable for dealing with features with different binding times. Although an aspect-based technique supposed to implement flexible binding times in a modular and convenient way [4] was introduced, we found some problems with respect to modularity, leading us to ask if aspects provide binding times flexibility without compromising the features modularity.

In this way, this paper reports an exploratory study that investigates whether the most popular AOP language (AspectJ) provides modularity when implementing features with flexible binding times. By modularity, we mean it is easy to maintain products with different binding times and feeders and there is no code duplication.

To assess the modularity provided by AspectJ, we conducted the study by implementing features with flexible binding times in two product lines. For each implementation, we discuss the advantages and disadvantages aiming to answer the question of this paper. We started with a simple product line. Some problems were found, but after refactoring the code, they seemed to be solved. The same did not happen for a real product line, so that our results suggest that the answer depends on the features peculiarities, such as size and heterogeneity. Answering the question represents the contribution of the paper, being useful to guide developers towards using AspectJ or not in the flexible binding time context.

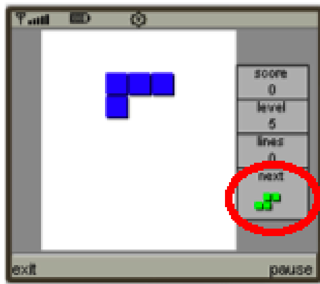
The rest of the paper is structured as follows. Section II presents the toy product line used to initiate our study. Because the binding time problems seemed to be solved in the toy example, we decided to analyze the binding time implementations in a real product line (Section III). Next, Section IV presents the related work. Last but not least, Section V tries to answer the question of the paper and concludes it.

II. Toy example

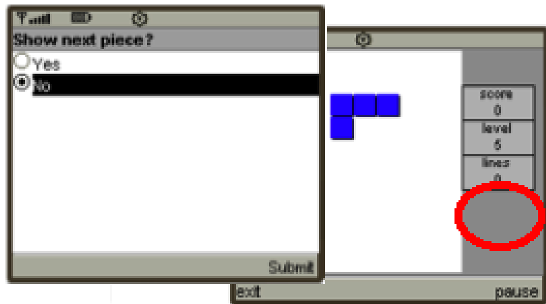
We begin our work focusing on the question of this paper: does AspectJ provide modularity when implementing features with flexible binding times? In order to investigate this, we have implemented features with different binding times using AspectJ to assess their modularity. The first

feature was extracted from a toy example, a Tetris J2ME game[1].

The feature analyzed consists of a next piece box. Such a box is responsible for showing the next piece that is supposed to be dropped into the game. Because this feature is optional, it is possible to generate games with/without the box. However, since we are concerned about binding times, it is important to note that we can generate three products: (i) without the next piece box; (ii) with the next piece box with no possibility to disable it; and (iii) with the next piece box with the possibility to enable/disable it. Notice that the binding time of (ii) is static, whereas (iii) is dynamic (see Figures 1(a) and 1(b)).



(a) Static next piece box.



(b) Dynamic next piece box.

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

After presenting the next piece box, we now explore two implementations of this feature using AspectJ. Each implementation deals with the different binding times presented. The first one is simple but has some problems. The second one provides a more elegant design, solving the problems of the first implementation.

A. Edicts

The next piece box feature consists of one class (*NextPieceBox*) and three code snippets within the *TetrisCanvas* class. Thus, we extracted its code from the *TetrisCanvas* class to edicts. Edicts is a technique to implement binding time flexibility of features in a modular and convenient way [4]. Edicts make possible to choose between static and dynamic binding times by using design patterns and aspects. Patterns[2] encapsulate the variation points whereas edicts (aspects) set the binding times of the features.

Using this approach, one aspect is responsible for the static binding time, whereas the other one implements the dynamic binding time. Games without the next piece box feature should not include neither the *NextPieceBox* class nor any aspect. The dynamic aspect also implements the feeder. In this paper, we consider a feeder as *a mechanism (or a set of mechanisms) responsible for providing information whether the feature should be enabled or not at runtime*. Examples of feeders are screens plus user input, configuration files, sensors, and so forth. In the tetris context, our feeder is a screen plus the user input, because depending on the user choice, the next piece box is displayed or not.

Listing 1 shows the *DynamicNextPieceBox* aspect. This aspect implements not only the next piece box code (Lines 4 to 22), but also the feeder (Lines 24 to 30), which consists of a screen displayed as soon as the game starts. The screen asks the user if he/she wants to enable the next piece box feature. If the answer is “Yes”, the next piece box code within the aspect must be executed. Otherwise, it must not.

We found some problems with this implementation. The first one is the *if* statements scattered throughout the aspect. For this particular example, three *if* statements might not be a problem. But when considering larger features with many advices, this scattering starts to be harmful. Notice that all advices must be encompassed by an *if* statement. If not, problems may occur. For example, if we remove the *if* statement of Lines 12 and 19 and the user does not enable the feature, the *nextPieceBox* attribute (Line 2) will be null and a *NullPointerException* would happen in the advices of Lines 10 and 18.

Listing 1. Next piece box with the dynamic binding time.

```
1  public privileged aspect DynamicNextPieceBox {
2    private NextPieceBox nextPieceBox;
3
4    after(TetrisCanvas canvas): nextPiece(canvas) {
5      if (getUserChoice()) {
6        nextPieceBox = new NextPieceBox(...);
7      }
8    }
9
10   after(Graphics g, TetrisCanvas canvas):
11       infoBoxes(g, canvas) {
12     if (getUserChoice()) {
13       ... nextPieceBox.setPieceType(
14         canvas.game.getNextPieceType()); ...
15     }
16   }
17
18   after(Graphics g): paintOnce(g) {
19     if (getUserChoice()) {
20       ... nextPieceBox.paint(g); ...
21     }
22   }
23
24   before(Command command, TetrisMidlet midlet):
```

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

[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.

```
25        commandAction(command, midlet) {
26      if (command == submitUserChoice) {
27        ... userChoice = nextPieceList.getString(
28          nextPieceList.getSelectedIndex()); ...
29      }
30    }
31
32    private boolean getUserChoice() {
33      return userChoice.equals("Yes");
34    }
35  }
```

According to the Edicts technique, a new aspect *StaticNextPieceBox* should be created for the static binding time. As the feature is now statically defined, we removed the feeder code as well as the *if* statements because they do not make sense for the static binding time. Nevertheless, the next piece box code is duplicated in both aspects, *DynamicNextPieceBox* and *StaticNextPieceBox*, resulting in a more serious problem since code maintenance becomes time consuming and error-prone.

Such a duplication problem gets worse if we must generate products that use other feeders. For example, suppose that a player is going to lose a game. Dynamically, the system could identify this situation and activate the next piece box feature aiming to help the player. This feeder looks like a sensor and has a completely different implementation when compared to the screen plus user input. Thus, another aspect must be created for the dynamic binding time using another feeder, duplicating even more the code.

B. Aspect Inheritance

Because the Edicts implementation did not result in a good design, we started to reason about a programming language construct able to provide us code reuse aiming to remove the scattering/duplication aforementioned. In this way, our second implementation relies on aspect inheritance through the abstract pointcuts definition.

Listing 2 shows the *NextPieceBox* abstract aspect. This aspect is responsible for implementing the next piece box feature and defining the abstract pointcut feeder (Line 2). The feeder pointcut is composed with the existing other pointcuts (Lines 4, 7, and 10), so that they will only be accomplished depending on the feeder pointcut. Again, we have scattering, specially when the aspect has many other pointcuts.

Listing 2. Abstract feeder and the next piece box feature.

```
1  public privileged abstract aspect NextPieceBox {
2    abstract pointcut feeder();
3
4    pointcut nextPiece(TetrisCanvas canvas):
5        ... && feeder();
6
7    pointcut infoBoxes(Graphics g, TetrisCanvas canvas):
8        ... && feeder();
9
10   pointcut paintOnce(Graphics g, TetrisCanvas canvas):
11       ... && feeder();
12
13   // The same three advices of Listing 1.
14   // However, there is no if statements here.
15 }
```

Now, we have to define the concrete feeders. Listing 3 shows two aspects responsible for this task. The *DynamicNextPieceBox* aspect implements the feeder code (the same screen of the Edicts implementation) and makes the feeder pointcut concrete, which returns *true* if the user wants to enable the feature. The *StaticNextPieceBox* aspect does not need to implement any feeder. It just assigns the feeder to be always *true*.

Listing 3. Extending NextPieceBox to define the concrete feeder pointcut.

```
1  public privileged aspect DynamicNextPieceBox
2      extends NextPiece {
3
4    pointcut feeder() : if (userChoice.equals("Yes"));
5
6    // Feeder code...
7  }
8  public privileged aspect StaticNextPieceBox
9      extends NextPiece {
10
11   pointcut feeder() : if (true);
12 }
```

By analyzing this implementation, we concluded that the most serious problem (code duplication) was solved: the next piece box code is not duplicated. In addition, new feeders just need to extend the *NextPieceBox* aspect and define the concrete feeder. Nevertheless, the *StaticNextPieceBox* aspect only exists to define something that makes no sense: the static binding time does not need any feeder.

Although this implementation improved the flexible binding time design, we decided to go deeper in such analysis. We believed that a small feature like the next piece box could hide some problems likely to occur in real systems. Thus, we extracted two features from a real system to aspects to analyze if the binding time flexibility remains.

III. Real example

In this section, we present two features we extracted from a real system named Freemind [5]. Freemind is an open source system used to construct mind maps. Mind maps are diagrams used to organize, structure, and classify ideas, being useful for brainstorm sections, making decisions, studies etc. Figure 2 illustrates a mind map in Freemind. It organizes the upcoming AOP events and their respective location and deadline for submitting papers. As showed in Figure 2, the map nodes may contain icons and clouds. For example, we used a cloud in the LA-WASP 2009 deadline to alert us it is coming up. Icons and clouds represent the features we extracted in this work.

Both features 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. Moreover, the information about the icons and clouds are saved in the file that represent the map. When loading this file, the system displays the correct icons/clouds in the correct nodes of the map. Besides these basic functionalities, both features are related to other concerns like "*Exporting*

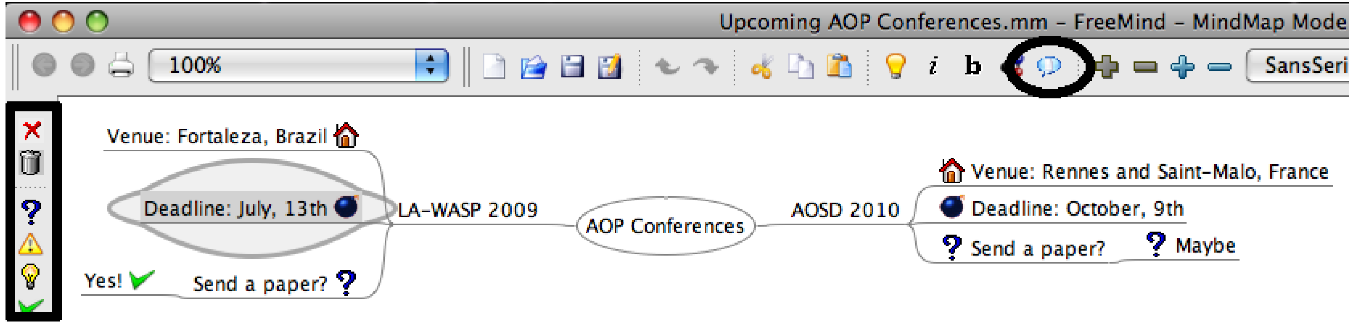


Figure 2. Mind map constructed in Freemind.

icons to html", being possible to export the map to HTML gathering its icons.

Thereby, to modularize the feature using AOP, it seems to be sound to create some aspects instead of only one. For instance, one aspect may contain icons/clouds GUI code whereas another is responsible for adding/remove them to/from the map. In summary, each aspect would be responsible for a concern of the features. For modularity reasons, we used this approach instead of creating a huge aspect to modularize the whole feature.

After studying the code of both features, we started to extract them to aspects. Since the features are heterogeneous and fine-grained, we found the same problems reported in [6]. For example, 21 hook methods were created to expose some join points and local variables, decreasing the design of some classes. Further, the known fragile pointcuts [7] problem has raised as well. Table I shows some numbers of each feature. Together, they represent 5.7% of the Freemind total LOC.

	Icons	Clouds
Hook methods	9	12
Affected classes	23	12
Classes	17	16
Aspects	8	6
Lines of Code (LOC)	2359 (3,1%)	2035 (2,6%)

Table I
SOME STATISTICS OF THE FREEMIND FEATURES.

As mentioned, we decided to implement both features using more than one aspect. To implement them with binding time flexibility, we decided to use the Aspect Inheritance approach because of the code reuse provided. However, we observed that this design does not scale. Because there is more than one aspect implementing icons/clouds, it is not possible to use inheritance, since AspectJ does not provide multiple inheritance. Nevertheless, one may implement the inheritance of Listings 2 and 3 for just one aspect. In the dynamic subaspect, say *DynamicIcons*, he/she may implement the feeder and create a method *getUserChoice*, which is called by the other aspects through the *aspectOf* method to know if the feature should be executed in those aspects. However, this solution is not suitable, because the scattering problem arises again in those aspects - *if (DynamicIcons.aspectOf().getUserChoice())*, and feature's code are duplicated in both static (without the *if* statement) and dynamic aspects, like in the Edicts design.

Therefore, we discarded this implementation and another one was taken into consideration. This new implementation relies on the *adviceexecution* pointcut (Listing 4). This solution seems to be suitable because there is no code duplication and the *if* statements are not scattered: the task of verifying the feeder is localized in this aspect, which matches all advices of aspects within the *freemind.icons* package. This solution may work for small features that use just *before* and *after* advices, but may not for bigger ones. Because the features icons/clouds are implemented using *around* advices that return not only *void* but also other objects types, when the *userChoice* is different of "Yes", this advice returns *null*, causing a *NullPointerException* in those icons/clouds advices. One may ask if we can use *void* instead of *Object* to remove the *return null* statement. Unfortunately, the *around* advices of icons/clouds do not compile because some of them do not use void but rather other object types. Problems related to the safety of the AspectJ type system are discussed in more detail elsewhere [8].

Listing 4. Adviceexecution pointcut.

```
1  Object around(): adviceexecution()
2      && within(freemind.icons.*)
3      && !within(AdvExecAspect){
4
5    if (userChoice.equals("Yes")) {
6      return proceed();
7    }
8    return null;
9  }
10
11 // Feeder code...
```

IV. RELATED WORK

Besides Edicts, we point out other related work. The first one [9] considers conditional compilation as a technique to implement flexible binding times in real systems such as operating systems. Just like our work, the developer would

decide if the feature must be present in the product as well as its binding time (see the example below). The work claims that conditional compilation is not very elegant and for more complex variation points, the situation becomes even worse.

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

CaesarJ [10] provides a mechanism to deploy aspects dynamically. If the aspect has the *deployed* reserved word in its signature, the aspect is statically deployed. Otherwise, it must be instantiated and deployed to starts working. Therefore, for dynamic features, the aspect that implements the feeder (*DynamicNextPieceBox*, Listing 5) should have the *deployed* word whereas the aspect responsible for the feature code (*NextPieceBox*) is deployed as long as the *userChoice* is "Yes". However, for products with static binding time, it is necessary to put the *deployed* word in the *NextPieceBox* aspect. For features that require many aspects, such task is error-prone, so that we need an additional technique like conditional compilation to insert/remove such a word.

Listing 5. Deployment mechanism of CaesarJ.

```
1  public cclass NextPieceBox {
2    ...
3  }
4
5  public deployed cclass DynamicNextPieceBox {
6    ...
7     if (userChoice.equals("Yes")) {
8       deploy new NextPieceBox();
9     }
10    ...
11 }
```

What the next related work [11] claims is that the actual AspectJ-like languages do not provide a mechanism to control the aspects scope dynamically. In other words, restricting the join points that an aspect should act is achieved not elegantly: conditions to the pointcut definitions should be introduced, sacrificing the reuse potential of aspects, as we showed. 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. The work proposes a model for restricting aspects scoping dynamically and we intend to use it as future work.

V. Concluding remarks

This paper presented a study to assess whether AspectJ provides modularity when implementing features with flexible binding times. According to our study, if AspectJ is applied in small features with few heterogeneity so that *around* advices are not required, the answer may be yes. For features implemented using just the *before* and *after* advices, the *adviceexecution* pointcut is useful for avoiding scattering of the *if* statement. Besides, this approach does not produce code duplication, even when adding new feeders.

However, if AspectJ should be used in big (requiring many aspects) and fine-grained features with a high degree of heterogeneity, our implementations lead us to answer no. The Edicts implementation duplicates code, meaning that it does not provide modularity at all. The Aspect Inheritance approach does not work properly with more than one aspect and the *adviceexecution* approach does not work when *around* advices are used to implement the features (and they really are, see Berkeley DB [6]). We believe that the problems get worse when implementing different features types like alternative (XOR) and OR and intertype declarations of methods are necessary. In this case, the developer must guarantee that calls to these methods (which might be inside or outside the aspect) are encompassed by the feeder to avoid runtime crashes.

In this paper, we provided some evidences supporting the use of AspectJ and others not supporting it. We have to analyze in deep other feature types and other AspectJ idioms like intertypes and *declare parents* to improve our assessment and consequently the answer to the paper's question. We intend to address these situations as future work.

It is important to note that our focus in this paper was only on AspectJ. Although another mechanisms like program transformations could easily add the *if* statements of the feeders, we did not address such implementations in this paper. Comparing mechanisms for implementing features with flexible binding times should be considered as more future work.

Acknowledgment

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 SPG[3] members for feedback and fruitful discussions about this paper.

References

[1] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "2nd international workshop on dynamic software product lines dspl 2008," in *Proceedings of the 2008 12th International Software Product Line Conference (SPLC'08)*. Washington, DC, USA: IEEE Computer Society, 2008, p. 381.

[2] M. Anastasopoulos and C. Gacek, "Implementing Product Line Variabilities," in *Proceedings of the 2001 Symposium on Software Reusability (SSR'01)*. New York, NY, USA: ACM Press, 2001, pp. 109–117.

[3] http://www.cin.ufpe.br/spg

[3] M. Ribeiro and P. Borba, "Improving Guidance when Restructuring Variabilities in Software Product Lines," in *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*. Washington, DC, USA: IEEE Computer Society, March 2009, pp. 79–88.

[4] 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.

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

[6] 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.

[7] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, "Information Hiding Interfaces for Aspect-Oriented Design," in *Proceedings of the 10th European Software Engineering Conference (ESEC'05/FSE'05)*. New York, NY, USA: ACM Press, 2005, pp. 166–175.

[8] B. D. Fraine, M. Südholt, and V. Jonckers, "Strongaspectj: flexible and safe pointcut/advice bindings," in *Proceedings of the 7th international conference on Aspect-oriented software development (AOSD'08)*. New York, NY, USA: ACM, 2008, pp. 60–71.

[9] E. Utrecht 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.

[10] 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.

[11] 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.