Declaring Static Crosscutting Dependencies in AspectJ

Alberto Costa Neto[1 2], Vander Alves[1], Paulo Borba[1]

[1]Informatics Center – Federal University of Pernambuco
Caixa Postal 7851, 50740-540 – Recife, PE, Brazil

[2]Department of Statistics and Computer Science – University of Sergipe
49.100-00 – São Cristóvão, SE, Brazil.

{acn,vra,phmb}@cin.ufpe.br

Abstract. *Aspect-Oriented Programming (AOP) is considered a promising approach for Software Product Line (SPL) implementation. In this paper we present the problem of dependency in inter-type method declarations as well a proposal of two new constructs to AspectJ interfaces: introduces and declares. These constructs can be used to declare and check dependency between base code and aspects in such a way to support separate development.*

1. Introduction

Aspect-oriented languages support the modular definition of concerns that are generally spread throughout the system and tangled with core features. These are called crosscutting concerns and their separation promotes the construction of a modular system, avoiding code tangling and scattering [6].

Filman and Friedman [4] proposed a concept in aspect-oriented design called *obliviousness*. They advocate that designers of base functionality do not need to anticipate or design the code to be advised by aspects. This concept leads designers to focus firstly on base functionalities and later on the crosscutting concerns design. A recent study [10] reports case studies showing that obliviousness makes it difficult or impossible to directly write the necessary pointcuts to intercept the join points and execute appropriate code in advices. It also proposes an approach that requires establishing *design rules* between base code and aspect designers.

We identified, in a J2ME Game software product line (SPL) implemented using AspectJ [2], the importance of declaring dependencies between base code and aspects caused by inter-type method and attribute declarations. These dependencies, when unplanned and uncontrolled, make developers (base and aspects) suffer from evolution problems. Additionally, these dependencies usually hinder the separated development of base and aspect code, since one depends on the other to compile.

This paper has two main contributions:

- Expose the problem of static crosscutting dependencies between classes and aspects (Section 2).

- Propose an extension to Java interfaces adding two constructs (`introduces` and `declares`) aiming at explicitly declaring those dependencies (Section 3).

We compare our work with others in Section 4. Finally, we present concluding remarks and future work in Section 5.

2. Problem

After constructing a Mobile Game SPL using an extractive approach by applying a series of aspect-oriented refactorings[2], it was noticed that classes and aspects presented some degree of mutual dependency, making difficult to evolve both independently. These dependencies were categorized with respect to their kind. In the next sections they are presented in more detail, using examples extracted from the SPL.

2.1. Classes depending on introduced class members

The first kind of dependency identified was of classes with relation to methods and attributes introduced by an aspect. In fact, these methods might have different implementations according to each specific family requirement.

The SPL core classes call some of the introduced methods directly at the appropriated points in code, requiring that some aspect introduces them during class compilation. This situation is different from a call made within an advice, since the method called is declared by the proper aspect via inter-type declaration.

The piece of code below shows calls to two introduced methods: `paintBeforeScrolling` and `paintBeforeScrolling` within method `paint` (defined in core class `GameScreen`).

```
public void paint(Graphics g) {
   Enemy myEnemy = null;
   int i = 0;
   int j = 0;
   g.setClip(0, 0, Resources.CANVAS_WIDTH, Resources.CANVAS_HEIGHT);
   paintBeforeScrolling(g);
   if (this.scroll.isScrolling) {
      paintScrolling(g);
   } else if(!isGameOver){
   ...
```

These methods have different behavior in each platform (Nokia S40, Nokia S60 and Motorola T720, for example) requiring a different inter-type method declaration and an aspect by platform. For brevity, only the Nokia S40 implementations are presented below.

```
public void GameScreen.paintBeforeScrolling(Graphics g) {
   if (this.isToClearAll) {
      this.scroll.reset();
      g.setColor(0);
      g.fillRect(0,0,Resources.CANVAS_WIDTH,Resources.CANVAS_HEIGHT);
      this.isToClearAll = false;
   }
}
```

```
public void GameScreen.paintScrolling(Graphics g) {
   if (MainCanvas.frame % 2 == 0) {
      this.scroll.paint(g);
   }
   this.drawArrow(g);
   if (!this.scroll.isScrolling) {
      this.aux = 255;
      MainCanvas.frame = 1;
   }
}
```

2.2. Aspects depending on class members

Another kind of dependency identified was of aspects with respect to attributes and methods referred within an inter-type method declaration. These class members can be referred by any aspect, including the private ones when an aspect is `privileged`. Any change to class interface may break aspects already written, since the class developer does not know which of its members is being used by aspects.

This situation is demonstrated in the piece of code below. The red boxes highlight some of the members that aspects assume to exist in the class.

```
public void Fire.update () {
   frame++;
   if (super.isVisible()){
      if (this.getY() < -40 || this.getY() > Resources.CANVAS_HEIGHT){
        super.setVisible(false);
        this.setXspeed(0);
        this.setYspeed(0);
      }
      super.setY(getY() + getYspeed());
      super.setX(getX() + getXspeed());
   }
}
```

The inter-type declaration above introduces a method called `update` in class `Fire`. It is important to notice that in order to develop the aspect, it's necessary to have access to the class `Fire`. It's difficult to develop separately classes and aspects because the latter requires the existence of the first to compile.

2.3. Using Java interfaces

We evaluated how Java interfaces could be used to reduce or eliminate those dependencies found. They were useful at some degree but were not able to completely solve the problems presented in sections 2.1 and 2.2.

The evaluated approach can be summarized in the following steps:

1. Define an interface containing the method signatures required by a class. These methods must be introduced by some aspect via inter-type method declarations;
2. Add the interface name to the classes implements clause;
3. Introduce the methods implementations in the interfaces via inter-type declarations.

Using this approach, it's possible to express which methods are required by a class. Also, it's possible to the aspect define a method introduction without knowing in advance the class.

The example below shows an interface called `ScreenPainter` that declares the two methods called (step 1) inside the `GameScreen`'s `paint` method. `GameScreen` class implements `ScreenPainter` (step 2). The `S40` aspect defines two inter-type method declarations required by classes (`GameScreen` in this example) that implement the interface `ScreenPainter` (step 3).

```
public interface ScreenPainter {
   void paintBeforeScrolling (Graphics g);
   void paintScrolling (Graphics g);
}
public class GameScreen implements ScreenPainter {}
public aspect S40 {
   public ScreenPainter.paintBeforeScrolling (Graphics g) {
      // Omitted Implementation
   }
   public ScreenPainter.paintScrolling(Graphics g) {
      // Omitted Implementation
   }
}
```

One possible improvement is defining an interface that lists all methods that classes must provide to aspects (see `Drawable` interface below). Classes can explicitly implement the interface or let aspects using `declare parents` to define which classes implement the interface. This allows declaring and checking which methods aspects can call as well as checking if classes provide the required interface.

```
public interface Drawable {
   int getX();
   int getY();
   void setX(int x);
   void setY(int y);
   void setXspeed(int xSpeed);
   void setYspeed(int ySpeed);
   void setVisible(boolean visible);
}
```

This approach, regardless of requiring no extension to AspectJ, has some important limitations that are explained below:

1. It's impossible to compile separately classes without an aspect that provides the methods declared in the interfaces;
2. Aspects can be compiled separately from classes in the presence of the interfaces, since the methods are introduced in the interfaces and not in the classes. On the other hand, since aspects do not know the target class, they can only refer to methods declared in the interface, what is not very useful.
3. Another limitation is that, besides methods, aspects need to refer to attributes in order to implement the inter-type method declarations and they can not be declared in interfaces (except as constants).

3. Solution

Considering the restrictions presented in Section 2.3, it's proposed in this Section an extension to Java interfaces to declare explicitly the dependencies presented in Sections 2.1 and 2.2.

The extension consists of adding two constructs to common Java interfaces, named **introduces** and **declares**. The first one is explained in Section 3.1 and the latter in Section 3.2.

3.1. Introduces

The `introduces` construct provides a way of exposing which members (methods and attributes) are expected to be introduced by an aspect implementing the interface. Classes implementing the interface can refer to these members as if they were defined by the proper class or inherited.

The syntax of `introduces` for attributes and methods is presented below:

```
introduces <modifiers> <type> <attribute-name>
introduces <modifiers> <type> <method-name>([<params-list>])
```

Where `modifiers` does not include the visibility modifier (`public`, `protected` and `private`). In summary, the syntax is similar to attribute and method declarations but preceded by `introduces` keyword.

Another difference from Java interfaces is that they must be present in both classes and aspects implements clause. This allows checking if an aspect that declares that implements the interface is in fact providing what is required by the `introduces`. From the classes point of view, it's possible to assume that the methods exist and use them as if they were declared in the proper class or inherited.

The piece of code below shows an example of `introduces`.

```
public interface ScreenPainter {

   introduces static int frameCount;
   introduces void paintBeforeScrolling(Graphics);
   introduces void paintScrolling (Graphics);

}
```

Above, the interface `ScreenPainter` could be defined to deal with the example problem presented in Section 2.1. It defines three `introduces`. The first one defines a `static int` attribute called `frameCount`. The second and third ones define the methods called `paintBeforeScrolling` and `paintScrolling` that must be introduced by aspects that implement the interface and can be used by classes that also implement the interface.

One of the purposes of declaring these introduces is being capable of using what is defined in the interface regardless of existence of aspects. As a consequence it should be possible to compile classes without aspects.

3.2. Declares

The `declares` construct exposes to aspects (that implement an interface) what methods and attributes are guaranteed to exist in the classes. As a consequence, it's possible to write, within inter-type method declarations, commands that manipulate these members, differently from the approach based on traditional Java interfaces discussed in Section 2.3.

The syntax of `declares` is presented below:

```
declares <modifiers> <type> <attribute-name>
declares <modifiers> <type> <method-name>([<params-list>])
```

Where `modifiers` does not include the visibility modifier (`public`, `protected` and `private`). This syntax is also similar to attribute and method declarations but preceded by the `declares` keyword.

```
public interface Drawable {
   declares int frame;
   declares int getX();
   declares int getY();
   declares void setX(int);
   declares void setY(int);
   declares void setXspeed(int);
   declares void setYspeed(int);
   declares void setVisible(boolean);
}
```

The interface `Drawable` above addresses the problem presented as example in Section 2.2. It uses a `declares` to guarantee that class `Fire` (shown in Section 2.2) has the attribute `frame` and also the methods `getX`, `getY`, `setX`, `setY`, `setXspeed`, `setYspeed` and `setVisible`.

The use of `declares` permits writing aspect's inter-type declarations decoupled from the target class, since any required member is listed on the interface.

4. Related Work

Open Modules [1] introduces a strong form of encapsulating join points occurring inside a module. It permits defining an interface composed by set of pointcuts that can be advised by clients. Any other join point that occurs inside the module is protected from external advising. Another work [9] proposes implementing Open Modules as an extension to AspectJ, using the AspectBench compiler [3].

Larochele et al [8] have proposed a mechanism, called join point encapsulation, which aims to prevent selected join points from being modified by aspects. They extend the AspectJ language to support a `restrict advice` that prevents the interception to specific join points.

Our work is complementary to those aforementioned since they attack the problem of encapsulation of join points, while our proposal attacks the static crosscutting dependency problem. Neither of them mentions or solve the problem arose in this paper.

Sullivan et al [10] proposes the specification of crosscutting interfaces (XPIs) to isolate aspect design from base code design and vice-versa through abstraction of crosscutting behavior. Griswold et al show how to represent XPIs as syntactic constructs [5]. These works are related to ours since we also want to isolate base classes and aspect design. In this paper we attack from a static perspective and they are concentrated on the dynamic perspective. Another difference is that they do not propose extensions to AspectJ an try to enforce their design rules when possible using existing AspectJ constructs.

Aspect-Aware Interfaces (AAI) [7] proposes an approach to modularity and modular reasoning in AOP. An AAI presents the dependencies between classes and aspects in bidirectional form, using annotations in both code. It permits better reasoning about the existing dependencies, but does not offer a way of avoiding or controlling such dependencies. Tools like AJDT offer similar information through the Crosscut References view. In fact, it could be used do define and maintain the AAI's. AAI differs from our approach since it does not treat the dependencies from classes to aspects and vice-versa found when using inter-type declarations. We propose an extension to Java interfaces to declare these dependencies.

5. Conclusions and Future Work

In this paper we expose the necessity of declaring static dependencies between classes and aspects in the presence of ITD's. We present two new constructs (`introduces` and `declares`) to Java interfaces that allow exposing and checking these static dependencies between classes and aspects.

We are planning to implement the constructs proposed in this paper using the extensible AspectJ compiler abc [3]. After that, we intend to evaluate this implementation in the SPL that motivated this work.

The addition of constructs `introduces` and `declares` is not enough to guarantee separate development in the presence of aspects with *pointcuts* and *advices*. We are currently working on others language extensions that encompass this problem.

6. Acknowledgments

We gratefully acknowledge our industrial partner, Meantime Mobile Creations, for granting us access to the game SPL. We would like to thank CNPq and Capes, Brazilian research funding agencies, for partially supporting this work.

References

[1] Jonathan Aldrich. **Open Modules: Modular Reasoning about Advice.** In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 3586 of LNCS, pages 144-168. Springer, 2005.

[2] Vander Alves, Pedro Matos, Leonardo Cole, Paulo Borba, Geber Ramalho. **Extracting and Evolving Mobile Games Product Lines**. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 70-81, September 2005. Springer-Verlag

[3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotáak, Ondrej Lhotáak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. **abc: An extensible AspectJ compiler**. In *Aspect-Oriented Software Development (AOSD)*, pages 87-98. ACM Press, 2005.

[4] Robert E. Filman, Daniel P. Friedman. **Aspect-oriented programming is quantification and obliviousness**. In *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, 2005.

[5] William G. Griswold, Macneil Shonle, Kevin Sullivan, Yuanyuan Song, Nishit Tewari, Yuanfang Cai, Hridesh Rajan. **Modular Software Design with Crosscutting Interfaces**, *IEEE Software*, Special Issue on Aspect-Oriented Programming, January 2006.

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

[7] Gregor Kiczales, Mira Mezini. **Aspect-oriented programming and modular reasoning**. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49-58, New York, NY, USA, 2005. ACM Press.

[8] David Larochelle, Karl Scheidt, Kevin Sullivanet. **Join Point Encapsulation**, *Proc. Workshop Software Eng. Properties of Languages for Aspect Technologies (SPLAT)*, 2003.

[9] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, Ganesh Sittampalam. **Adding Open Modules to AspectJ**. In *Aspect-Oriented Software Development (AOSD)*, pages 39-50, ACM Press, 2006.

[10] K. Sullivan, et al. **Information Hiding Interfaces for Aspect-Oriented Design**, In *Proceedings of ESEC/FSE´2005*, pp.166-175, Lisbon, Portugal, September 5-9, 2005.