Motivations for Framework-based AOP

Bruno De Fraine, Wim Vanderperren, Davy Suvée
System and Software Engineering Lab (SSEL)
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
{bdefrain,wvdperre,dsuvee}@vub.ac.be

ABSTRACT

In comparison to language-based AOP approaches, aspect-oriented frameworks have the inherent advantage of a better integration with the existing development process. In this paper, we discuss some of the less evident advantages of framework-based approaches for general-purpose AOP: ability to build programmatic abstractions, flexibility and dynamic nature, a better integration between aspects and the base application, and openness and customizability. These benefits are part of the design criteria of the new Eco approach.

1. INTRODUCTION

General-purpose aspect-oriented approaches can be roughly divided in two categories: *framework-based approaches* and *language-based approaches*. Framework-based approaches, such as JBoss [3], PROSE [14], AspectS [8], Reflex [17] and AspectWerkz [2], introduce an aspect-oriented framework where the specification and deployment of the aspectual behavior is organized entirely in the base programming language. Language-based approaches on the other hand, such as AspectJ [10], JAsCo [16] and CaesarJ [12], employ a dedicated language or language extension for developing aspects. The obvious advantages of a framework approach are the compatibility with the existing development process and the easier acceptance of the approach. Dedicated aspect languages on the other hand, have the advantage of being more concise, intuitive and straightforward in their descriptions.

In this position paper, we discuss some of the less evident advantages of framework-based AOP approaches. To this end, we analyze the possibilities of programmatic abstraction, flexible application of aspectual behavior, and integration with the base application. Note however that the claimed benefits in this discussion are of a theoretical nature: we believe these advantages are applicable to approaches built entirely according to a framework philosophy. They might however not apply to some of the current AOP frameworks, e.g. because they employ a dedicated language for the specification of pointcuts instead of a representation in the base language. The arguments of this paper are part of the design considerations of the Eco approach [4] that enables all of the advantages discussed here.

2. PROGRAMMATIC ABSTRACTION

It is an inherent advantage of framework-based approaches that the abstraction mechanisms of the base language (like procedures and class hierarchies) are available for the specification of aspectual behavior. In dedicated language solutions this is not automatically the case, and often the language entities cannot be called first-class. In such a case, there are for example no suitable means to avoid the duplication between similar (or largely similar) entities. We analyze this situation for three common entity types in aspect-oriented approaches: *advices*, *pointcuts* and *deployment constructs* to bind an advice to a pointcut.

Advices

In contrast to the other entity types, advice behavior is generally specified in the base language, even in language-based approaches. As most approaches also allow advice methods to be inherited similar to regular methods, there is no immediate difference in the possibilities for abstraction compared to AOP frameworks. Meta-facilities only accessible to advices (such as the `proceed()` method) are an exception however. In most languages, deferring the implementation of a complete *around* advice to an auxiliary method (to share the implementation with another *around* advice, for example) is not possible as the proceed method is not available outside of the advice. The only work-around for this situation is to wrap the proceed method in an anonymous class, as shown in listing 1 (a similar technique is also employed in the *worker object creation* pattern [11] to construct objects that represent executable methods). Framework-based approaches have no such obstacles hindering possible abstractions, as they only employ base language features in the advice definition. An *around* advice method typically takes in a framework object through which it can invoke the original join point behavior (e.g. the `Invocation` type in JBoss).

Pointcuts

The majority of AOP approaches, both framework and language-based, use a dedicated language for the specification of pointcuts. Although this allows for a concise and convenient notation, the options for programmatic abstraction

```
interface Proceedable {
  public void doProceed();
}

public aspect MyAspect {
  public void auxMethod(Proceedable p) {
    // Can invoke p.doProceed()
  }

  void around(): somePointcut() {
    auxMethod(new Proceedable() {
      public void doProceed() { proceed(); }
    });
  }
}
```

Listing 1: Work-around to abstract complete around advice (including `proceed()`) in an auxiliary method.

```
Pointcut getSettersPC(Class theclass) {
  return new ExecutionPC(
    void.class,                 // return type
    theclass,                   // containing class
    "set*",                     // method name
    new Class[] {Object.class} // argument type(s)
  );
}
```

Listing 2: Generic procedure returning a pointcut to select the setters of a given class.

are limited. In AspectJ's pointcut language for example, it is only possible to combine pointcuts using boolean operators. We believe that a representation of pointcuts in the base language allows for much stronger abstractions and improves code reusability. For example, a common idiom is to write a pointcut to select the execution of the *setters* of a class named `Order` as `execution(void Order.set*(*))`. In a Java representation of the same pointcut model, the pointcut designator to select the execution of methods could be represented as a class `ExecutionPC`[1]. This class can be used to directly define the pointcut, but it also becomes possible to build a generic procedure to select the execution of the *setters* of a given class, as illustrated in listing 2. Using this procedure, the pointcut can simply be defined as `getSettersPC(Order.class)`. This abstraction not only avoids the code duplication between *setter* pointcuts, it also hides the implementation details of selecting *setters* as methods whose name starts with "set" and which have a void return type. Should we later decide on a more robust way to construct *setter* pointcuts (cfr. [7]), then we only need to change the procedure `getSettersPC` to update all pointcuts. Currently only advanced pointcut languages [7, 13] are able to offer such abstraction features.

Deployment constructs

When pointcuts and advices are independently defined entities, an additional deployment step is required to bind these two parts. Aspect-oriented approaches use various techniques to support this: subaspects (e.g. AspectJ), separate connector entities (e.g. JAsCo), XML configuration files and code annotations (e.g. JBoss and AspectWerkz). When a run-time API is offered for the purpose of deployment, the possibility of building abstractions over the deployment process is seldom considered (JBoss, for example, offers its deployment API with the objective of enabling run-time recomposition). Nevertheless, we observe a large amount of code duplication in the deployment process, especially when it includes the installation of aspect ordering and combination strategies. The possibility of building reusable deployment strategies could reduce the amount of boilerplate code and improve software maintainability.

[1]Note that the proposed Java representation still supports most features of dedicated pointcut languages, but changes the notation in which pointcuts are written.

3. NO GLOBAL STATIC ASPECTS

By default, aspect-oriented programming employs a view where aspects specify their behavior as global rules over the structure of an application (*"whenever method `Foo()` is called, write a log entry"*). In AspectJ for example, aspects by default have a singleton instance that exists for the entire time of execution and that cuts across the entire program. This corresponds with a model of global and static aspects, i.e. aspects that apply to all objects in the base application, at all times. However, a trend towards dynamic aspects [6] and association aspects [15] has proven this view to be too simplistic. To a certain extent, we notice that the process of making aspects dynamic and non-global involves giving them abilities of standard classes (multiple instances, custom run-time construction,...). In case of an object-oriented base language, a framework-based approach already organizes all aspectual behavior in terms of classes and objects. As such, it can natively meet these requirements.

More concretely, part of the term *dynamic aspects* refers to the ability to recompose aspectual behavior at run-time. With regard to this concern, the value of a run-time API for aspect deployment, as offered by framework-based approaches, has previously been recognized (e.g. in JBoss and AspectS). *Association aspects* on the other hand, refers to aspects with different instances for different objects or object groups. From [15], we learn that a substantial part of the proposed solution for association aspects consists in replacing AspectJ's implicit instantiation by an explicit aspect instantiation that does not force argumentless constructors. This is, in other words, the standard instantiation model for classes from the base language, which aspect-oriented frameworks normally already employ.

Although aspect-oriented frameworks require further measures to fully support the association aspects from [15], their ability to flexibly create aspect instances in the application is valuable in general to realize more powerful, dynamic aspects. This is illustrated in listing 3, where we organize custom tracing of user sessions in an application. Upon login, we deploy two instances of a session tracing aspect which output to different places: one to the personal log of the user, the other to the application log. Although implicit instantiation can be instructed to create multiple instances (e.g. by using the `perthis` aspect modifier), it cannot cope with this level of flexibility.

4. BASE/ASPECT INTEGRATION

Asymmetric AOP approaches make a strong distinction between the *base* application that implements the main or

```
1 void login(User u) {
2   Session s = new Session(u);
3   new SessionTraceAspect(s, u.getLog());
4   new SessionTraceAspect(s, Application.getLog());
5   ...
6 }
```

Listing 3: Flexible, explicit instantiation of aspects.

business concerns, and *aspects* that implement crosscutting concerns. However, a level of integration is needed between these two parts as aspects may require parameters from the base application for their operation (parameters that are not directly available at the intercepted join point). Concrete examples of such information could be the handle of the application's log file in case of a tracing aspect, or the observers to notify in case of an aspect that intercepts changes to an observed subject. Similarly, the base application may also need to read out results from any of the aspects.

In approaches with implicit instantiation, a first solution for this problem could be that the aspect implements additional aspectual behavior to capture (or expose) the required information. This is not straightforward and error-prone: e.g. in case of a required parameter, one has to verify that in all cases this capture has succeeded before the main aspectual behavior of the aspect is triggered. A second solution involves making information available for static lookup: either the application makes the desired objects available through static fields or methods, or the aspects are retrieved by the base application through AspectJ's static `aspectOf()` construct (or an equivalent in other approaches), after which the base application *injects* (or retrieves) the information in the aspects. Solutions involving a static lookup impose a restriction as the information must be addressable with a unique key. For example, in case of more complex instantiations such as the tracing behavior of the previous section, it becomes increasingly difficult to employ `aspectOf()`: suppose two aspect instances could be implicitly created for a session object, by what key would we address each of them to pass the relevant log output?

The usage of explicit instantiation and plain code deployment in framework-based approaches offers a more elegant solution for this integration. As shown in listing 3, the aspectual behavior can be deployed from the point where the parameter information is normally available in the base code, and the explicit instantiation enables to obtain a reference to the aspect instance. Finally, notice that base/aspect integration for aspect deployment does not necessarily violate the principle of *obliviousness* [5]. Although the deployment code is aware of the aspects and has to be modified in order to unplug the aspectual behavior, the classes and methods in the base application that are advised by the aspects are not, and their implementation does not need to be tangled with other concerns.

5. OPEN AND CUSTOMIZABLE

A natural advantage of frameworks is that they are more open to adaptation and extension than language-based approaches. This is because frameworks offer their facilities as libraries, on the same level as the base application. As such, frameworks can provide extension points where clients can pass in their own functions, classes or objects to influence the different parts of the approach and to customize the aspectual behavior. An example of this is presented in [9], where it is shown how AspectS can be extended with reasonable ease to support new features such as process-specific aspects.

Unfortunately, most aspect-oriented frameworks do not seem to pursue this goal of customizability and rather focus on offering an equivalent of standard language-based approaches with the advantage that the developer can employ the base programming language. For instance, aspect composition is typically handled in a way that cannot be extended nor customized beyond some predefined solutions. This definitely underemploys the possible advantages of framework-based approaches.

6. CONCLUSIONS

In this position paper we present and motivate the general advantages of a framework-based AOP approach versus a language-based AOP approach. We identify improvements related to (1) ability to build programmatic abstractions over all parts of the aspect specification, (2) flexibility and dynamic nature of aspect instantiation, (3) integration between aspects and the base application and (4) openness and customizability.

As a final note, we observe that the support for flexible, abstractable and dynamic aspects is not exclusive to framework-based approaches. Most of the discussed features could be included into a language-based approach, similar to how e.g. AspectJ supports abstraction through aspect inheritance. However, this process involves making the aspect language largely similar to the base programming language. It is therefore useful to take an opposite view: starting from a framework-based approach, which features from language-based approaches are still lacking, and can they be added? In the field of Java AOP frameworks for example, static type-safety for aspects was one of the main features only offered by language-based approaches. It is therefore one of the concerns that the Eco approach specifically addresses in [4].

7. ACKNOWLEDGMENTS

Bruno De Fraine and Davy Suvée are supported by a doctoral scholarship from the Institute for the promotion of Innovation by Science and Technology in Flanders (IWT).

8. REFERENCES

[1] M. Akşit, editor. *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003).* ACM Press, Mar. 2003.

[2] J. Bonér and A. Vasseur. AspectWerkz: simple, high-performant, dynamic, lightweight and powerful AOP for Java. Home page at `http://aspectwerkz.codehaus.org/`, 2004.

[3] B. Burke et al. JBoss Aspect-Oriented Programming. Home page at `http://www.jboss.org/products/aop`, 2004.

[4] B. De Fraine, W. Vanderperren, and D. Suvée. Eco: A flexible, open and type-safe framework for aspect-oriented programming. Technical Report SSEL 01/2006/a, Vrije Universiteit Brussel, Jan. 2006. `http://ssel.vub.ac.be/files/defraine-eco06a.pdf`.

[5] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In P. Tarr, L. Bergmans, M. Griss, and H. Ossher, editors, *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, Oct. 2000.

[6] R. E. Filman, M. Haupt, and R. Hirschfeld, editors. *Dynamic Aspects Workshop*, Mar. 2005.

[7] K. Gybels and J. Brichau. Arranging language features for pattern-based crosscuts. In Akşit [1], pages 60–69.

[8] R. Hirschfeld. Aspect-oriented programming with AspectS. In M. Akşit and M. Mezini, editors, *Net.Object Days 2002*, Oct. 2002.

[9] R. Hirschfeld and P. Costanza. Extending advice activation in aspects. In K. Gybels, M. D'Hondt, I. Nagy, and R. Douence, editors, *2nd European Interactive Workshop on Aspects in Software (EIWAS'05)*, Sept. 2005.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[11] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[12] K. Ostermann and M. Mezini. Conquering aspects with Caesar. In Akşit [1], pages 90–99.

[13] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *Proceedings of ECOOP 2005*, 2005.

[14] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 141–147. ACM Press, Apr. 2002.

[15] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*, pages 16–25. ACM Press, Mar. 2004.

[16] D. Suvée and W. Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In Akşit [1], pages 21–29.

[17] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. R. Lowry, editors, *Generative Programming and Component Engineering, 4th International Conference (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188. Springer, Sept. 2005.