FacetS: First Class Entities for an Open Dynamic AOP Language

Alexandre Bergel
Distributed Systems Group
Trinity College
Dublin 2, Ireland
www.cs.tcd.ie/Alexandre.Bergel

ABSTRACT

This paper describes a new aspect language construct for Squeak, named FACETS. Aspects are completely integrated within the Squeak programming language and its environment. The innovations of FACETS are: (i) *traits* can be part of the pointcut definition, (ii) two scoping policies are available to share state among aspects and (iii) aspects are prototype-based.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1.5 [**Programming Languages**]: Object-oriented Programming

General Terms

Language, Design

Keywords

Aspect, traits, inheritance, state, prototype

1. INTRODUCTION

Aspect-oriented programming (AOP) [11] holds the promise of composing software out of orthogonal concerns. AOP promotes code insertion (*i.e.,* advices) at some particular locations in the source code (*i.e.,* join point shadow). Advices are executed when the control flow reaches a joint point shadow. Some dynamic requirements like pattern in the method call stack can also be specified.

Current AOP approaches suffer form two well know problems:

- Once weaved, aspects are "melted" in the base system. No discernment can therefore be made between the base system and the aspects. As a result, debugging and reverse engineering are greatly hampered.
- Most of AOP systems require a dedicated language different from the base language. This increases the amount of effort required to use aspect oriented techniques.

In this paper, we present FACETS, a new dynamic aspect system for which: aspects can be installed and removed at runtime, aspects are incrementally built in the *same* language as the host system. Code weaving is based on the use of wrapper. FACETS is currently implemented in Squeak [10, 18] benefiting from its reflective features. The contributions of this paper are:

- Traits can be part of a pointcut definition.
- Two scoping policies for sharing variables among aspects.
- Description of prototype based aspects.

Firstly, Section 2 provides a description FACETS, showing its main characteristics. Then, Section 3 presents a few points of the implementation. Section 5 raises some open questions and presents a future direction of FACETS. Finally, Section 6 concludes this paper.

2. FIRST CLASS ENTITIES IN AOP

FACETS is a new dynamic aspect system where aspects are incrementally and programmatically defined (in the same spirit as Steamloom [3]). Moreover, aspects can be dynamically installed and removed.

This section uses the scenario previously introduced by Hirschfeld [9] to illustrate the key points of FACETS. This example is based on introducing a trace over the whole graphic class hierarchy in Squeak. Methods named mouseEnter: are instrumented with some printing into the standard output stream. This method is triggered when the mouse cursor is moved over a widget area.

2.1 Incremental Construction

In our terminology, an *aspect* is a collection of advices. And an *advice* encapsulates the code instrumentation and specifies the location where this instrumentation should occur.

Advice. An advice is created by (i) instantiating the class Advice, and providing (ii) the instrumenting code, (iii) a list of pointcuts, and a (iv) qualification of the instrumentation (*i.e., before*, *after*, and *around*):

```
advice := Advice new.

advice code: [:wrapper |
   Transcript show: 'Invoke: ', wrapper selector,
      ' args: ', wrapper arguments printString,
```


Open Aspect Languages Workshop (AOSD) Bonn (Germany).

```
    ' receiver: ', wrapper receiver printString ;cr].

advice addPointcut: [:behavior :method |
  (behavior inheritsFrom: Morph) and: [ m = #mouseEnter: ]].

advice qualification: #before.
```

The code provided to the advice is a block (also considered as a function or a closure) which takes as parameter a wrapper containing runtime information for instance, the name of the invoked method (selector), the list of arguments (arguments) and the object receiver of the message (receiver).

Pointcut. A join point is an element of the language semantics that the aspect coordinates with. It traditionally identifies location within the source code of an application. A *pointcut* is a predicate used to identity join points.

In FACETS, a pointcut is represented as a function that takes as argument a behavior (*i.e.,* a class or a trait, discussed in Section 2.2) and a method name (method), and returns whether this particular method is a join point or not. In the example above, the pointcut [:behavior :method | (behavior inheritsFrom: Morph) and: [m = #mouseEnter:]] identifies subclasses of Morph (root of the graphical class hierarchy) for which the method mouseEnter: is defined. Note that in this version of FACETS variable access is not supported.

Classes in Squeak do not have constructor, but rather a method named initialize which is called when this class is instantiated. Object creation can therefore be advised by defining a pointcut for the method basicNew.

Several pointcuts can be added to an aspect. The corresponding advice is activated when one of the pointcuts is true. Note that the pointcut are static, meaning that they are evaluated at weaving time.

Aspect. Aspects are created by instantiating the class Aspect. Advices are then added to this instance. The advice described above is added into a new aspect:

```
aspect := Aspect new.
aspect addAdvice: advice.
```

Once instantiated, an aspect can be freely installed and removed at runtime.

2.2 Traits and Aspects

Traits are recognized for their potential in supporting better composition and reuse, hence their integration in newer versions of languages such as Perl 6, Squeak [10], Scala [16], Slate [17] and Fortress [6]. One important innovation of FACETS is to allow traits to be part of a pointcut definition.

Description of traits. The trait model is a new language construct proposed by Schärli *et al.* is an alternative to multiple inheritance [5]. Traits are essentially sets of methods that serve as the behavioral building block of classes and the primitive units of code reuse. Classes (and composite traits) are composed from a set of traits by specifying glue code which connects the traits together and accesses the necessary state. With this approach, classes retain their primary role as generators of instances, while traits are purely units of reuse. As with mixins, classes are organized in a single inheritance hierarchy, thus avoiding the key problems of multiple inheritance, but the incremental extensions which classes introduce to their superclasses are specified using one or more traits.

Pointcut designating traits. A pointcut is represented as a predicate taking a behaviour and a method name as parameters. This behaviour can either be a class or a trait. A trait can therefore be the subject of a join point definition. For instance, the following pointcut identifies the method named compile: in a trait used by the class Behavior:

```
advice addPointcut: [:behavior :method |
  (behavior isTrait and:
    [ behavior users includes: Behavior ]) and:
      [ method == #compile: ]].
```

Defining an aspect on a trait makes the methods of this trait (*i.e.,* defined methods and methods obtained from a trait composition) which are wrapped with an advice. When a trait is used by some classes, aspects defined on this trait are then defined on the using classes. The flattening property of the traits is "A non-overridden method in a trait has the same semantics as if it were implemented directly in the class." A consequence of this property is that if some aspects are defined on a trait, these aspects are also defined on the class which use this trait.

Advice and traits. Dynamic information related to the advice activation is available to the code contained in an advice. This information is provided to the code contained in the advice by means of the wrapper. A wrapper understands the message receiver. The result of it is the instance of the current receiver, *i.e.,* an instance of a class or a subclass that uses a trait.

2.3 Dynamic Installation and Removal

An aspect can be installed and removed by simply invoking the methods install and remove on the instance which describes this aspect. For instance, aspect install instruments all the methods mouseEnter: defined in each subclass of the class Morph. In the case of a multi-threaded environment, it might occur that a method is under execution when it is instrumented. In this case, the instrumentation has effect only for future invocations.

In a similar way, aspects can be removed by simply invoking the method remove on an aspect: aspect remove uninstall the aspect. All the instrumentation of the methods mouseEnter: are removed. If an instrumentation is under execution while an aspect is removed, only future invocation will not invoke the instrumentation.

Installation of an aspect is done by sending either the message installOn: or install to the instance of an aspect. The method installOn: takes as argument a list of potential classes that this aspect can be applied to. The method install assumes that all the classes in the system may potentially be part of a pointcut definition.

2.4 Properties

This section enumerates the properties of the FACETS aspect mechanism.

Global visibility. Whereas in Steamloom [3] an aspect can be thread local or instance specific, FACETS does not provide any scoping mechanism. Limiting the visibility of an aspect to a thread has the advantage of bounding the "responsibility" of aspect installation/removal only to the control flow orders it. In other words, the thread that installs/removes an aspect cannot impact other running threads. In FACETS, an aspect is globally visible, therefore installation and removal may affect the entire system.

Conflicts are forbidden. Composing aspects is a challenging task. Current approaches are based on specifying a proper order of aspect application [2] or various method instrumentation [12].

In the current version of FACETS, we deliberately left conflicts management for future work. Currently, we forbid a method to be instrumented more than once.

Aspect at runtime. Most of current AOP approaches "melt" an aspect into the code at weave-time. Code analysis in software reverse engineering is therefore hampered because the base code is "polluted" with aspect code. FACETS tackles this issue by keeping aspects distinct from the base code (in terms of source code and bytecode). Aspects and base system can be independent of the subject for analysis, even after being weaved.

Unique language. Traditionally, AOP systems use a dedicated language to specify pointcuts and advices. Applying AOP techniques therefore necessitates the knowledge of a language different from one use to develop the base system. FACETS uses the same language, Squeak [18], to write aspects and applications. Aspects are instantiation of plain standard classes, and pointcuts are block structures (similar to Scheme's closure and Java's anonymous inner classes).

Application of aspects on classes which use traits. In most common implementation of aspects (*e.g.,* AspectJ [2]), an aspect *fully* defines *what* and *where* advices have to be applied to.

With FACETS, the user classes of this trait are not known when an aspect is defined on a trait. An aspect can later on be applied to a class by making this class use the trait. The responsibility of designing the location of an aspect is broadened.

Prototype-based aspect. In FACETS an aspect is defined as an instance of the class Aspect or one of subclass of Aspect. Because a class can define state (*i.e.,* instance variables), an aspect therefore encapsulates state that is accessible only to the advices contained in this aspect. Moreover, an aspect can be instantiated more than once in FACETS. As in Squeak a class can have *shared class variables*, variables can be shared among all the instance of a subclass of Aspect.

In FACETS, two scoping are available to share state: (i) *aspect instance scoped*, variables are shared among the advices of a particular aspect instance, and (ii) *aspect family scoped*, variables shared among instances of an aspect class (and its subclass).

This has to be put in contrast with AspectJ-like approaches where an aspect applies the singleton pattern [7]: with AspectJ, an aspect can be instantiated only once. It means that AspectJ supports only *aspect instance scope* to share variable.

3. IMPLEMENTATION

Contrary to AOP systems like AspectJ [2], FACETS does not use bytecode manipulation to instrument methods, but rather uses built-in reflective features of Squeak. The instrumentation or a method consists in generating a wrapper for it.

Most of structural elements of the object model are first class entities in Squeak: a class, a method and a method dictionary is accessible as any standard plain object. In Squeak, a *class* has, among some other attributes, a name, a superclass, and a reference to a method dictionary. The set of methods defined by a class is defined by a *method dictionary*. A method dictionary contains a list of associations $\langle$ *methods name, compiled methods* $\rangle$. And finally, a *compiled method* is a list of bytecode, directly executable by the virtual machine.

In FACETS, a method instrumentation consists in generating a wrapper. This wrapper is inserted between the method dictionary and the compiled method.

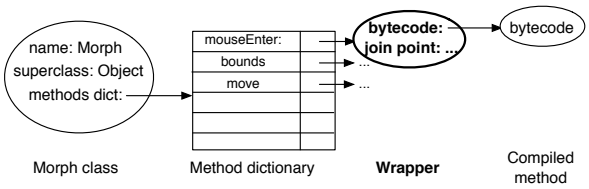

Figure 1: The method mouseEnter: is instrumented.

Figure 1 depicts the instrumentation of the method mouseEnter: on the class Morph. The wrapper containing a reference to the join point is inserted between the method dictionary and the compiled method. On sending the message mouseEnter: an instance of Morph invokes the wrapper. This instance has then the control on the message sent and can trigger the compiled method according to the join point. Note that this implementation is similar to the method wrapper [4]. The difference is

4. RELATED WORK

This section put FACETS in perspective compared to relevant works. FACETS is the only aspect mechanism to support traits, we therefore do not mention this difference for each enumerated work.

AspectS. The first AOP system designed for Squeak is AspectS [9]. AspectS and FACETS differ regarding the aspect construction. With AspectS a new aspect is created by subclassing the class Aspect. An advice is then associated with a method.

With FACETS, an aspect is created by instantiating the class Aspect and by providing pointcuts and advices. FACETS supports incremental definition of aspects.

Steamloom. Steamloom [3, 8] is a highly optimized extension of IBM's Jikes virtual machine supporting dynamic join points. Aspects can be dynamically deployed and removed according to three kinds of visibility: global, thread-local, and instance-local. Steamloom therefore implicitly supports a notion of context as a unit of computation (*i.e.,* thread) and as a structural runtime entity (*i.e.,* object). Incremental definition of aspects is also supported.

Steamloom mainly differs from FACETS in the implementation and the kind of pointcut supported . Steamloom necessitates a dedicated virtual machine, where FACETS runs on the classical Squeak VM. Moreover FACETS allows a trait to be part of the pointcut definition. An aspect is applicable to a trait and to a class.

Classpects. The AOP system Classpects [15] proposes a unification between classes and aspects. Each structural element (*i.e.,* aspect, join point, advice, ...) are first class entities. However, it does not provide any dynamic ability for installing and removing aspects at execution time.

Reflex. In the context of Java, Reflex [19] provides building blocks for facilitating the implementation of different aspect-oriented languages so that it is easier to experiment with new AOP concepts and languages, and it is also possible to compose aspects written in different AOP languages. It is built around a flexible intermediate model, derived from reflection, of (point)cuts, links, and metaobjects, to be used as an intermediate target for the implementation of aspect-oriented languages.

CaesarJ. Aspects, packages and classes are unified in CaesarJ [1]

under a single construct. Similar to Steamloom [3, 8], aspects deployment can either be global or thread local. An aspect can also be deployed and removed at runtime. CaesarJ does not provide first class entity pointcuts, therefore activation of an aspect is determined by join points (*i.e.,* a result of a pointcuts language expression).

5. FUTURE WORK AND FURTHER QUESTIONS

The current version of FACETS described in this paper is an early prototype of a new vision for AOP. This section lists few relevant points to be discussed at the workshop.

Conflicts and aspects. Aspects in the term of AspectJ [2] cannot be used to define unanticipated changes. The main reason is the lack of efficient visibility policies. Aspects may conflict with each other, and aspects may conflict with the base system (*e.g.,* with the introduction mechanism of AspectJ).

Hyper/J [13] does a better job than AspectJ regarding conflict management. Applying an aspect (*i.e.,* a feature according to the Hyper/J terminology) creates a copy of the base system (named a *hypermodule*), leaving the original base system intact.

These issues are well know. Ostermann and Kniesel summarized these issues in their previous work [14]. Aspects are an excellent mechanism to define crosscutting concerns when these are foreseen, however they are poor when extensions with a well limited impact have to be defined.

Regarding this topic, some more general questions are: *What would a good visibility mechanism be for AOP? Why not avoid conflict instead of dealing with its resolution?*

Aspect based language kernel. As far as we are aware, none of the existing AOP languages place aspects at the core of their object model. Whereas classes and aspects have similarities [15], classes are not defined in term of aspects. Whereas Smalltalk, ObjVLisp, CLOS have a clean meta model, none of the aspect languages provide a clean kernel. *What would be an aspect based reflective kernel of a language ?*

6. CONCLUSION

This paper describes a new AOP systems for the Squeak programming language. By having as first class entities structural AOP elements like aspect, advice and pointcut, FACETS allows for (i) incremental definition of aspects, (ii) dynamic aspect installation and removal, (iii) non invasive weaving, leaving the base code intact and (iv) uses the same language to describe aspects and the base system. The current version of FACETS is available on `www.cs-.tcd.ie/Alexandre.Bergel/FacetS.zip`.

Current challenges are illustrated with some open questions regarding impact and visibility of aspects and also regarding an aspect based language kernel.

AOPD is a new methodology that brings better modularity into software. It also benefits from a deep knowledge within the scientific community. However it has little support from industries. One reason for this is that AOP is currently not placed at the heart of software development. This is exactly the goal of FACETS, making AOP closer to developer.

Acknowledgments. We gratefully acknowledge the financial support of the Science Foundation Ireland and Lero — the Irish Software Engineering Research Centre.

We also would like to thanks Serena Fritsch, Eamonn Dillon and Jenny Munnelly for their helpful readings and comments.

7. REFERENCES

[1] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of caesarj. *Transactions on Aspect-Oriented Software Development*, 2006. To appear.

[2] AspectJ home page. http://eclipse.org/aspectj/.

[3] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[4] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998. method wrappers.

[5] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems*, Mar. 2006. To appear.

[6] The fortress language specification. research.sun.com/projects/plrg/fortress0618.pdf.

[7] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In O. Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[8] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An execution layer for aspect-oriented programming languages. In *Proceedings VEE 2005*. ACM Press, June 2005.

[9] R. Hirschfeld. AspectS – Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in LNCS, pages 216–232. Springer, 2003.

[10] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, Nov. 1997.

[11] G. Kiczales. Aspect-oriented programming: A position paper from the Xerox PARC aspect-oriented programming project. In M. Muehlhauser, editor, *Special Issues in Object-Oriented Programming*. Dpunkt Verlag, 1996.

[12] I. Nagy, L. Bergmans, and M. Aksit. Composing aspects at shared join points. In A. P. Robert Hirschfeld, Ryszard Kowalczyk and M. Weske, editors, *Proceedings of International Conference NetObjectDays, NODe2005*, volume P-69 of *Lecture Notes in Informatics*, Erfurt, Germany, Sept. 005. Gesellschaft für Informatik (GI).

[13] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM Press, 2000.

[14] K. Ostermann and G. Kniesel. Independent extensibility – an open challenge for aspectj and hyper/j. In *Proceedings of Aspects and Dimensions of Concern Workshop*, 2000.

[15] H. Rajan and K. J. Sullivan. Classpects: Unifying aspects- and object-oriented language design. In *Proceedings*

International Conference on Software Engineering (ICSE 2005), pages 59–68, 2005.

[16] Scala home page. http://lamp.epfl.ch/scala/.

[17] Slate. http://slate.tunes.org.

[18] Squeak home page. http://www.squeak.org/.

[19] É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *LNCS*, Tallin, Estonia, sep 2005.