Traits Programming with AspectJ

Simon Denier

Traits Programming with AspectJ

Simon Denier

Obasco Project (EMN - INRIA)
Ecole des Mines de Nantes
4, rue Alfred Kastler, BP 20722
F-44307 Nantes cedex 3
Simon.Denier@emn.fr

ABSTRACT. *Traits as defined by Schärli, Ducasse* et al. *allow for the explicit handling of a meaningful set of methods. This article presents an attempt to map the trait model from Smalltalk to the Java language. We use AspectJ introduction mechanism to do this. Thus we enlighten purposes of locality and reusability shared by traits and structural AOP.*

RÉSUMÉ. *Les traits tels que définis par Schärli, Ducasse* et al. *permettent la manipulation explicite d'un ensemble de méthodes sémantiquement corrélées. Cet article présente une tentative de portage du modèle des traits sous Smalltalk vers le langage Java. Nous réalisons ce portage à l'aide du mécanisme d'introduction d'AspectJ. Ainsi nous illustrons les objectifs de localité et de réutilisabilité communs aux traits et aux aspects structurels.*

KEYWORDS: *traits, aspects, reusability, composition, Java, AspectJ.*

MOTS-CLÉS : *traits, aspects, réutilisabilité, composition, Java, AspectJ.*

RSTI - L'objet – 11/2005. Programmation par aspects, pages 69 to 86

1. Introduction

Traits for class-based languages, as introduced in Squeak by (Schärli *et al.*, 2003), are lightweight entities which serve as basic building blocks for classes, primitive units of code reuse and elements of class structuration (Ducasse *et al.*, 2005). Traits come as sets of methods without state. The purpose of traits is reusability: they address some limits of inheritance in object-oriented languages. In particular they are well-suitable in single-inheritance language like Smalltalk, where they avoid code duplication. The compositional model supporting traits seamlessly integrates methods into classes: it allows a fine, explicit control on the composition of traits by means of operators. Programming style and practice (Black *et al.*, 2003) suggests that traits best fit as small sets of related methods.

As another single inheritance language, Java does not have such a mechanism. Taking a look at AspectJ – a language experimenting aspects in Java (Kiczales *et al.*, 2001) – we notice one special feature: inter-type declaration, formerly known as introduction. The inter-type declaration mechanism offers a subset of structural reflection, which allows for extension/modification of classes (with methods, fields, interfaces, etc.) at compile-time. Inter-type declaration is described as a means to modularize crosscutting structure. In particular, basic examples (available on `http://eclipse.org/aspectj/`) seem to pursue quite the same goals as traits: to provide general, possibly reusable, features (*i.e.* states and methods) in a modular way. One purpose of our work is to explore the relationship between traits and a structural subset of AOP.

The core section of this article describes our experiment to build a trait-like entity in Java thanks to AspectJ, with respect to the Squeak traits model. Starting with the need for structuration of trait behaviors, we will see how the reusability concern is handled, how we can express requirement and how we build a class from a composition of traits. The goal of this experiment however is not to propose an implementation for traits in Java, but rather to push existing mechanisms to an end to get a view of their limits. So we will finally see how conflicts are partially resolved by class delegation.

The remainder of this article is structured as follows. We give an overview of Smalltalk traits in section 2. Section 3 exposes our experiment to emulate traits properties in Java with AspectJ introduction. We then discuss in section 4 some questions on delegation, conflicts and static typing. Section 5 concludes with some thoughts on modules and aspects.

2. Overview of Smalltalk Traits

Inheritance is considered as a fundamental way to promote reusability in object-oriented languages. However the properties and constraints introduced by current mechanisms do not allow to achieve full capabilities (see (Schärli *et al.*, 2003) for a discussion on failures of different inheritance models).

In fact inheritance and in particular multiple inheritance suffers from a dual conception of class: as a generator of instances, a class must be completely descriptive; but as a unit of code reuse, it should stay small, focused on its subject matter. There is a discontinuity between the level at which we actually express and allow reuse (class, mixin) and the level at which we really want reuse (methods).

So traits propose to address part of the reusability concern at a finer-grained level, by providing sets of behaviors – each set being soundful and intended for generic use – a flattening yet seamless compositional model and some mechanisms for conflict resolution. Traits are well-suited for single inheritance languages, for which they avoid code duplication. However traits currently do not address state reuse, as no general solution is known for state conflict resolution [1].

2.1. *Anatomy of a Trait*

A trait is a stateless entity which provides a set of methods. Provided methods may be parameterized by some other methods, which are referred to as required. Required methods are only declared in the trait. In particular, required methods can be used to specify accessors for states/properties that the trait needs but does not define. Figure 1 shows a schematic view of a trait, as introduced by Schärli *et al*.

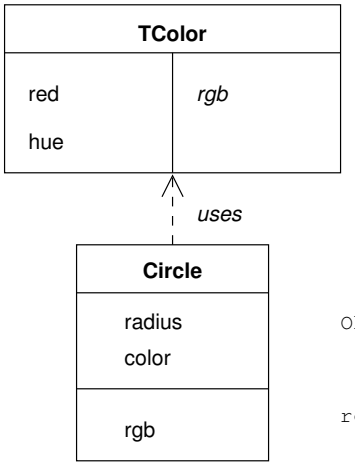
2.2. *Using Traits*

Class definition in Smalltalk has been enhanced with a `use` field, which allows for the declaration of expected traits. The `use` field cardinality is multiple, as a class may use many traits; the order of composition is irrelevant. A class eager to use a trait must provide implementations for the required methods by local means or with the help of other traits. Traits can use other traits in the same way a class does: they are called composite traits. The flattening property, one of the most distinguishing property of traits, happens at the class level: trait methods behave as if they were directly defined in the class. It follows that a class enhanced with traits can also be seen without the traits syntax. The use feature and the flattening property allow for the primitive composition mechanism, referred to as a sum. The equation below gives the point:

$$Class = Superclass + States + Traits + Glue$$

"Glue" is a vocable primarely covering methods that fulfill requirements from traits. Glue code is needed for a soundful composition of class definitions with traits. In Figure 1 the `Circle` class uses the `TColor` trait; the implementation of `rgb` is glue code.

1. We used version 0.5 of the traits prototype for Squeak during our experiments.




```
Trait named: #TColor uses:{}

red
     ^ (Color rgb: self rgb) red

hue
     ^ (Color rgb: self rgb) hue

rgb
     self requirement
```

```
Object subclass: #Circle
    uses: {TColor}
    instanceVariableNames: 'radius color'

rgb
     ^ color
```

Figure 1. *The trait `TColor` provides some methods shown in left column and requires the method in right column. The `Circle` class is enhanced with `TColor` behaviors by* using *the trait and providing a state and the required method. Equivalent Smalltalk code is given*

2.3. *Conflicts Resolution*

Smalltalk traits may conflict at composition time when providing identically named methods that do not originate from the same trait[2]. Two mechanisms help to resolve most of conflicts, whether in a class or in a composite trait: first methods defined by the compound entity (class or trait) have the property to override methods from used traits; second the use field can contain method excluding statements to suppress some trait methods from the compound. Another mechanism, aliasing, allows the access of overriden or deleted method by another name. Together these handle conflicts between provided methods by simply selecting one implementation among others, or by merging the implementations into a new one (see 3.4.1 for an illustration in Java).

2.4. *Guidelines for Traits Implementation*

From traits we retain the following ideas. These are the guidelines for our emulation in Java:

– ability to localize a set of (related) methods in a dedicated entity (and possibly reify a trait as a language element);

2. Then the diamond case is not considered a conflict.

– reusability, provided by modularity, independance from class and easiness of use;

– ability to express requirement (as abstract methods for example);

– ability to compose clearly and cleanly with classes;

– ability to have composite traits – create traits from traits;

– ability to detect and resolve conflicts in clean and meaningful ways.

3. Our Experiment: Emulating Traits in Java with AspectJ

Through this section, we target a `Circle` class which we want to enhance with a color property in a modular fashion, aside from the inheritance tree. We will gradually introduce AspectJ notions and their roles in trait simulation, then examine the example in consequence. Finally we will see how we enhance our model with some kinds of physics property. Introducing this property will cause a conflict with the color property which we will then resolve.

3.1. *Providing Methods in a Reusable Way*

Traits provide behaviors independently from the traditional hierarchy of classes: methods definition is localized in a separate entity targeting a concern to promote sharing and maintenance. Its usage must be independent from the primary inheritance mechanism, which first expresses a type family.

3.1.1. *Placing Methods in a Container*

We need to place the code in a container apart from the class hierarchy, and then make accessible the implementation inside the container in an existing class. We do this by means of AspectJ introduction mechanism: an aspect module can contain some identifiers and method code which will be introduced at weaving time in the chosen classes. Then, at runtime, a modified class behaves as if introduced methods were directly defined in it. So we have some flattening property as in Smalltalk traits. As shown in Figure 2, the aspect module `TColorRepository` introduce methods `getHue()` and `getRed()` in class `Circle`, as denoted by the special notation `Circle.getHue()` and `Circle.getRed()`.

In case the class already provides by itself a definition for a method, the compiler will not introduce the one from the aspect. Methods from class seem to have priority over introduced methods in AspectJ [3]. This is the same behavior as overriding trait methods in Smalltalk, which allows for the resolution of those conflict cases in a natural way. On the contrary, the AspectJ compiler is not able to dismiss the case

3. This is an experimental result. Although not described in the AspectJ documentation, this is an intuitive behavior.

when a method is both inherited from the superclass and introduced by an aspect: it declares this as a conflict. This conflict can still be resolved in the same way others are (see 3.4). This is a divergence from the Smalltalk flattening property, whose definition implies inherited methods are overriden by traits methods. We will discuss this in 4.2.

From the programmer point of view, the aspect acts as a container for the trait implementation but does not hold the trait identity. That is we can use the color trait unaware of the `TColorRepository` module. By convention, we call the aspect a repository and distinguish it with a name suffix. So our trait-like entities will each have their own `T<property>Repository`.

```
aspect TColorRepository {
   public float Circle.getHue() {
      float[] hsb = Color.RGBtoHSB(getRed(),
                                   getGreen(),
                                   getBlue(),
                                   null);
      return hsb[0]; }

   public int Circle.getRed() {
      return this.getRGB().getRed(); } // getRGB() is a requirement

   ... //same for getGreen(), getBlue()
}
```

Figure 2. *An aspect module as a repository for behavior implementation*

3.1.2. *Using and Reusing Methods from the Container*

So far we have localized the implementation but we need more to be able to reuse the methods code: we should not have any reference to classes that use the trait. Currently this is not the case as we do have a reference to `Circle` within `TColorRepository`. Suppose we want to use our trait in a `Rectangle` class: we have to introduce the reference in `TColorRepository` and, even worse, to duplicate the code as this syntax does not allow to introduce the same portion in multiple classes.

In fact, the problem happens because we implicitly instantiate the use relation between `Circle` and the trait inside `TColorRepository` (notice we have not changed the `Circle` class for the time being). But it is rather a responsibility of the class to explicitly express its need for the trait with a use relation. Then we also need a way to express the use relation in a class.

Fortunately, AspectJ's introduction allows *introduction in combination with an interface*. This means that AspectJ can declare a method as part of an interface and link it to the implementation in an aspect module. In Figure 3, we use the interface `TColor` as an identifier for the color property. The interface declares its methods

signatures: although it is not mandatory in AspectJ introduction, we gain in clarity by doing so. Then the `TColorRepository` module gives an implementation for each method in the `TColor` interface.

By this we throw away the reference to the `Circle` class and replace it with a kind of abtract reference, in total independance with the existing hierarchy. But this interface usage also gives the use relationship we need: any class which *implements* a trait interface is provided with the implementation in the repository, thanks to AspectJ introduction (Figure 4). As the implement relation coexists in Java with the extend relation, we can use a trait interface anywhere in a class hierarchy. So our trait is reusable. The key relation is that the interface stands for the trait identity.

```
interface TColor {  // Trait interface
   public int getRed();
   public float getHue();
   ...
}

aspect TColorRepository {  // Behaviors implementation
   public int TColor.getRed() { ...} // change Circle to TColor
   public float TColor.getHue() { ... } // change Circle to TColor
   ...
}
```

Figure 3. *A basic trait example. Notice how the name changes from `Circle` to `TColor` in the introduction syntax since Figure 2*

3.2. *Expressing Requirements*

Traits need to integrate their behaviors with classes they are applied to: they need some requirements from classes to interact with. Moreover, since states are not allowed in traits, classes have to provide the necessary structure to support traits. These requirements take the form of methods signatures, particularly accessors, which classes have to implement jointly with some states.

A Java interface, referred to as a requirement interface (Figure 4), is a natural way to express those: required methods take the form of declared methods in the interface which are simply not implemented by the aspect module. A class eager to use a trait then fullfills these requirements with “glue code”: it provides some implementation for the interface as well as the underlying structure (states) implied. In our emulation

we make the trait interface extends the requirement interface [4]. Moreover we have the compiler to check the implementation of required methods.

```
interface TColorRequirement {  // Trait requirement
   public Color getRGB();
   ...
}

interface TColor extends TColorRequirement {  // Trait interface
   public int getRed();
   public float getHue();
   ...
}

aspect TColorRepository {  // Behaviors implementation
   ...
}

class Circle implements TColor {
   private Color color; // State for requirement
   public Color getRGB(){ return color; }
   ...
}
```

Figure 4. *The complete `TColor` trait example with repository, interface and requirement. Then the `Circle` class provides some state and the required method `getRGB()` so as to implement the `TColor` trait and benefit from it*

3.3. *Composing Traits*

Traits target the reusability concern in a way similar to multiple inheritance. That is, a class should be able to use behaviors from multiple origins – its own corpus, from parent classes and from traits. So composition is a major feature of traits implementation. It should be carefully looked at as it is also a source of conflicts.

Composing a class from multiple traits is straightforward since interfaces in Java are not constrained by the single inheritance rule. So a class just has to declare implementing as many trait interfaces it wants, as long as this does not induce some conflicts.

4. The application of the introduction mechanism to interfaces is disturbing as it breaks the programmer's view of Java interfaces as a mean to enforce signatures. With AspectJ the role is reversed as interface becomes a mean to provide implementation. That is why we separate both concerns – contract enforcement by required methods and reusable code by provided methods – in two different interfaces.

Composing a trait from multiple traits – producing composite traits – is also straightforward as an interface could extend some others. Then a trait eager to reuse another one must extend the other's interface. By this, it can also provide some implementations for the used trait requirements; unprovided requirements are still being propagated by the interface mechanism.

By propagation of types, the AspectJ weaver recognizes all traits interfaces and introduces methods from the right repositories. In Figure 5, trait `TColor` is enhanced with trait `TEquality` which provides some identity predicates. However the syntax seems confusing as the same mechanism (extension of interface) allows for the propagation of requirement and for the composition of trait.

```
interface TEquality extends TEqualityRequirement { ... }

interface TColor extends TColorRequirement, TEquality { ... }
```

Figure 5. *An example of trait composition: trait `TColor` reuses `TEquality`*

3.4. *Resolving Conflicts*

Conflicts ask for a meaningful composition dealing with the class purpose. There is many ways to do this: it could be by separating, prioritizing, excluding, merging, encapsulating or cross-referencing features (Bracha *et al.*, 1992). Some of these solutions imply combinations of aliasing and exclusion as in Smalltalk while others need some kind of scope or module.

AspectJ can not manipulate methods at the grain aliasing and excluding operators ask for. To work around conflicts, we have to come back to delegation which supports many features – although in a verbose manner.

3.4.1. *Using Delegate Classes*

Class delegation is a way for a complex class to deal with its multiple behaviors in a modular way. Delegating means that these behaviors will actually be provided by smaller agregated classes. For the purpose of traits, a delegate class can be an intermediary level between a trait and the compound class: it comes with the minimal structure and requirement needed by the trait so as to be self-sufficient, but it does not realize the integration with others traits. It is still the class responsibility to do so.

There is many ways a class can work with delegates, according to its purpose: it can enable state sharing between delegates or, on the contrary, ask for a strict separation of state. It satisfies message sends to trait methods by itself or by forwarding to its delegates. Indeed, thanks to the *class over trait* priority rule (see 3.1.1), its self-

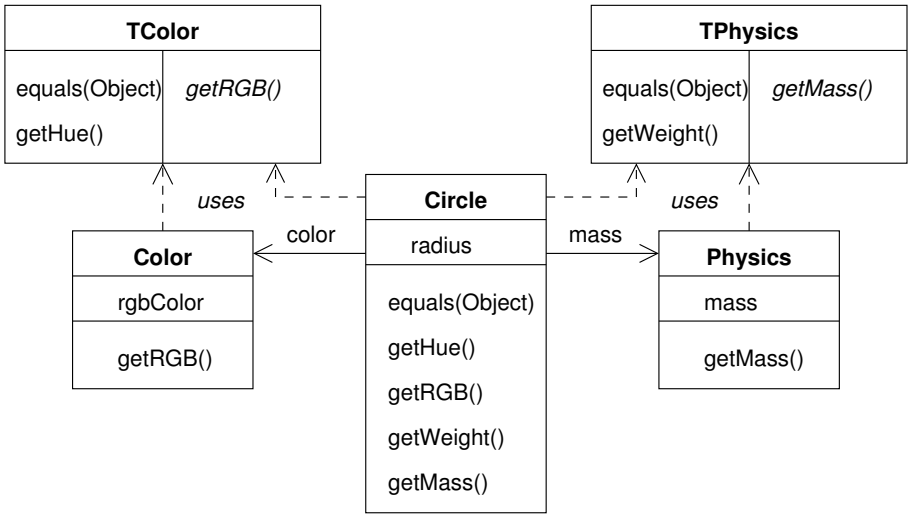


Figure 6. *Using simultaneously* `TPhysics` *and* `TColor` *traits is a case for conflict, as both provides* `equals` *method with their own meaning. For* `Circle` *class, the problem can be solved by containing each implementation in a delegate class and providing a meaningful interface, either delegating, merging, excluding behaviors from delegates*

contained implementation overrides those of traits, possibly bypassing conflicts. This offers many ways of resolving conflicts:

– selecting one behavior among others by delegating to it;

– merging delegate behaviors in the new delegating method;

– aliasing a behavior thanks to an indirection to a delegate.

In Figure 6, we enhance the `Circle` class with `TColor` and `TPhysics` traits. As both traits provide an `equals(Object)` method, each one targeting its own property, conflict arises. In Figure 7, we choose to merge both behaviors. We could also have excluded the color equality property from the default `equals` method, but still provided it via an alias, like in Figure 8.

Here are the guidelines for composing conflicting traits. First, each trait should be provided with a delegate class which implements the trait interface and satisfies structure and requirement. Then, for each trait, the compound class comes with a private instance variable of the associate delegate class. A delegate class can be designed to be sufficiently generic and reusable, or to simply fit the compound class purpose. The compound class can also implement all trait interfaces and, for each provided or required methods, must provide either a delegating method or one of the solution above.

```
class Color implements TColor { // Delegate class
   private java.awt.Color color;
   public java.awt.Color getRGB() { ... }
   ...
}

class Physics implements TPhysics { ... } // Delegate class

// Using class
class Circle implements TColor, TPhysics {
   private Color color; // trait TColor delegate
   private Physics physics; // trait TPhysics delegate

   // delegating method
   public int getHue() { this.color.getHue(); }
   ...

   // merging equals behavior
   public boolean equals(Object anObject) {
     return physics.equals(anObject)
         && color.equals(anObject); }
}
```

Figure 7. *A complete example of conflict resolution by delegation and merging, from Figure 6.* `getHue` *is a single behavior of* `TColor` *and hence needs only delegating to the* `Color` *instance.* `equals` *behaviors from* `TColor` *and* `TPhysics` *are merged*

So combining delegation with the previous methodology allows to:

– resolve name conflicts by merging, excluding, aliasing;

– provide a meaningful, carefully selected, interface for the compound class;

– restrain the execution scope of a trait to a delegate, thus disabling possible semantic mischieves (see 4.2);

– create either specific or generic delegate class, thus enabling for a set of reusable "glue" classes for traits.

But there is also some drawbacks from the use of delegation in our design. There is the cost of implementing the framework – delegate class and delegating behaviors – although most IDEs nowadays can do this in an automatic way. There is the cost of one indirection at runtime. However the main drawback comes from the breaking of flattening property. We discuss this issue in section 4.1.

```
// Using class
class Circle implements TColor, TPhysics {
   private Color color;
   private Physics physics;

   ...

   // excluding TColor.equals()
   public boolean equals(Object anObject) {
         return physics.equals(anObject); }

   // aliasing TColor.equals()
   public boolean colorEquals(Object anObject) {
         return color.equals(anObject); }
}
```

Figure 8. *An alternative to merging: first the `TPhysics equals` behavior is preferred to the `TColor` one as default behavior (or, the `TColor equals` is excluded); second, the `TColor equals` behavior is provided* via *an alias*

3.4.2. *Conflicts in Composite Traits*

Creating a composite trait actually means composing abstract methods in interfaces. So a conflict in a composite trait will not be detected until a class actually uses it and the weaver tries to introduce the different implementations. Thus the resolution of conflicts in a composite trait is postponed to classes using it. Moreover this began to be cumbersome when we have an hierarchy of composite traits, each possibly coming with conflicts.

The AspectJ compiler allows overriding in composite traits: methods provided by the composite override methods from inherited traits. But as we have no aliasing, we can not reuse the overriden methods: this clearly limits the practical interest for this use of overriding. One solution is to provide for each composite a generic delegate class. But then, dealing with both an hiearchy of traits and a set of delegate classes to reuse is cumbersome. Using delegates is definitely not as simple as traits are.

To conclude, we can not express all the semantics of composite traits in a modular way: resolving conflicts in composite traits is the point where our emulation fails.

4. Discussions

This section presents some thoughts on conceptual and practical issues dealing with the traits model as well as with our emulation. We highlight some questions con-

cerning delegation, conflicts or types – sometimes to give a provocative look towards Smalltalk model choices.

4.1. *Delegation: Drawbacks and Enhancement*

An obvious criticism of delegation is the cost and the noise introduced by delegating methods, which produce runtime overcost and confuse the programmer about class capabilities. However the main drawback, which is a conceptual one, is that the scope introduced by delegation breaks the flattening property. The scope allows for a clean separation of context and obvious semantics of behaviors (see 4.2.2). The flattening property allows for a quick and seamless integration of traits: for example one can fulfill a requirement from another in an oblivious way. Doing such a thing with delegation means having a reference in the delegate outside its scope. This implies a link to the current implementation and so a more fragile structure.

On the other hand, delegates offer the means to provide generic, reusable stateful traits; although they will not fit well everywhere, they are direct to implement, easy to understand and have obvious semantics. Notice that we are not arguing that traits should be delegate classes, but that each trait could come with its delegate class. This benefits the reusability concern.

4.2. *Cases for Conflicts*

Generally speaking a conflict arises when features with the same identity are combined along different paths. The identity is primarely defined by a name and could be refined by type or signature for a method. The path distinction is prone to discussion depending on whether or not a same origin make the distinction (diamond case).

4.2.1. *Conflicts Between Traits*

When composing multiple traits in the same class, the obvious case for conflict arises when provided methods with same name are picked up along different paths (case A in Figure 9). Since the order is irrelevant, the language can not choose between different implementations. However, if the origin is the same, then we could easily consider there is no conflict; if not, then a choice need to be made by the programmer, depending on the purpose of its class.

The problem is a bit different when it comes about required methods combined along different paths (case B in Figure 9). Due to a lack of semantics on requirements, the common hypothesis is to consider those identical as the same: then one implementation should fit for all. The right question is: does the concerned traits share the same requirements? By example, should they share a state or not? In case they should, how do we check that range constraints on type are compatible? Among solutions we can think of a scope notion or some kind of aliasing for requirement.

A last case (C in Figure 9) does not have to be considered as a conflict but rather as a potential pitfall for programmer. When a provided method from a trait matches a required method from another one, the first will silently satisfy the second at composition time. The operation could be oblivious to the programmer – this is different from the case where the programmer gives a "glue" method to fulfill a requirement. However, as both traits could have been developped in independance, nothing tells us that the implementation really fits the requirement. Thus a warning could be issued by the language to inform the programmer.

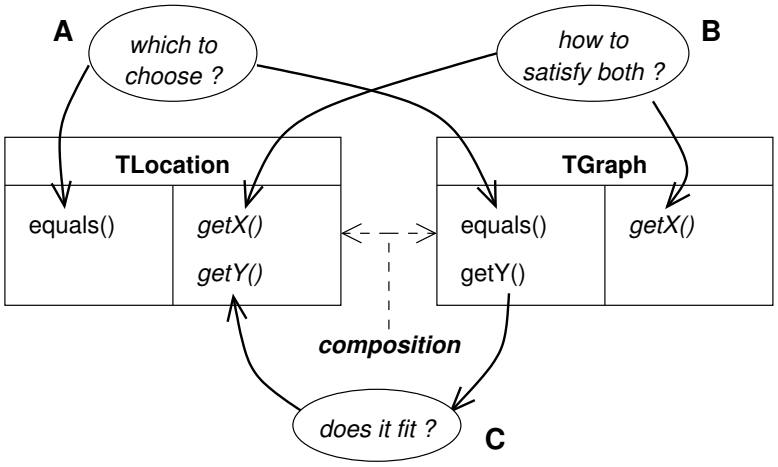


Figure 9. *Three cases for conflicts in trait composition. When combining together* `TGraph` *and* `TLocation`*, the programmer should dismiss conflicts and meaningless links by carefully examining the situation. (A) Which behavior to choose for* `equals`*? One of both or a merge of both or nothing? (B) Does* `TLocation` *and* `TGraph` *share the same requirement when asking for* `getX`*? (C) Is the* `getY` *implementation from* `TGraph` *convenient for the requirement of* `TLocation`*?*

4.2.2. *Stress between Classes and Traits*

The overriding property of class methods over trait methods allows one to resolve conflicts in a nice and simple way. However, its lack of scope can also break the implementation, as the overriding code will impact on the whole trait hierarchy used. Suppose a method provided by a trait is called in the trait subhierarchy. Then, if a class using this trait hierarchy overrides this method with its own implementation, the last one will prevail, even in place where the previous was appropriate.

Finally we consider the rule for conflicts between traits and the superclass. In the Smalltalk model, trait methods override ones from the superclass. Although this is a natural consequence of the flattening property, deciding which methods from a superclass or from a trait to choose depends on the class purpose. *A priori*, one could argue that superclass possibly best fits the particular concern addressed by the class

whereas a trait targets a generic yet broad concern. In AspectJ the choice is to declare this as a conflict.

4.2.3. *On Conflicts, Scope and Modules*

The use of scope to solve conflicts is a fundamental of module systems. However this notion operates at large scale and can be favourably enhanced by fine-grained operations like aliasing or excluding: this is the proposal from Bracha in its Jigsaw model (Bracha *et al.*, 1992) – but we must then give up some simplicity. A more recent alternative comes from Classboxes (Bergel *et al.*, 2005), a module system where local rebinding allow to narrow the impact of changes to local features – possibly helping with the problem above.

4.3. *Static Types Impact*

The implementation of traits in a statically typed language like Java raises new questions, on the way traits are intrinsically influenced by types but also on the way they impact on the existing type system. At first sight types can reduce the number of conflicts by distinguishing between signatures, in the same way they allowed overloading. However one should be aware of a pitfall already present in the type system of Java: when considering concrete methods, the language makes no distinction between those who only differ by their return type; but this difference exists when it comes about abstract methods. So one can be trapped in a pathological case where requirements differing by their return types can not be satisfied, as implementations conflict.

Another, more practical, question is if user-defined types are allowed in traits definition. For example, a method parameter could be defined by a dedicated interface. Thus the interface becomes part of the requirements: it is the responsibility of the trait user to define a class satisfying the parameter interface. It can enhance the expressivness of traits as well as disturbing its usability by asking "too much" requirements. Also the design of reusable traits could entail the use of the `Object` class as a generic type, which implies casting or exception handling. We can expect that generics in Java 5 will ease this situation.

At last, a fundamental problem arises when thinking of a trait itself as a type. Without trait typing, using a trait in Java would mean that trait methods are transformed to be plain members of their compound class and loose connection with the trait identity. For example a binary method `foo(T)` defined by a trait `T` will be later refined as `foo(B)` as a class `B` is using it. Clearly we lost some genericity in doing so, as a instance of class `C` using `T` could not be passed to `foo(B)`.

In our experiment, the trait identity is assimilated to an interface: so using a trait for a class means being of the trait type, in the same way as implementing an interface or extending a class. Therefore a `return this` statement in a trait method simply returns an object of the trait type. This also enables the direct definition of `foo(T)` in

class `B` and so an instance of class `C` can be passed: this is consistent with what we expect from polymorphism in Java.

Yet we have another problem considering the exclusion operator. Take a class C excluding from its interface a method of trait T: if we call this method on a T instance (T being the static type) which resolves in a C instance at runtime, the Java virtual machine will stop by throwing `NoSuchMethodError`. Indeed one purpose of the static type system is to guarantee at compilation time that any method call is correct. Our first assumption combined with the exclusion definition breaks the safety of the type system. The use of interfaces as trait identifier in our emulation, coming from AspectJ, is misleading.

We can think of two solutions to work around this problem. The first and easy one is simply to throw away the excluding operation. Indeed we can still resolve conflicts by overriding them [5]. But we lost some capabilities and ease of use by doing so. The second solution is to change the semantics of the exclusion: it defines a new type which is not any more compatible with the original trait type and its hierarchy. This can be interpreted as the fact that it defines a new anonymous trait before composition without the subtyping relation [6]. So there is a cost in using the exclusion operator which the programmer should be aware of. To push this thought to an end, we observe the same kind of duality which classes suffer with inheritance: classes play a role in implementing methods but they also play a role in defining a type, which primarely guarantees the interface.

5. Conclusion

The benefits in reusability, maintenance and readibility expected from traits are of high value for a mainstream language like Java. We enlighten in this article some key ideas for an implementation of traits. We illustrate those by an experiment with AspectJ introduction mechanism and interfaces, emulating some traits properties. However we did not achieve the full capabilities of traits regarding conflicts resolution, as we lack fine-grained operators. Even so we were able to get a good emulation by using delegation, providing both drawbacks and enhancements.

We also extend our thoughts on conflicts by showing how the flattening property can sometimes lead to a non-desirable behavior, and how a scope notion is useful in those cases. Finally we discuss with some thought experiments in Java the way static types impact the trait model and reciprocally. In particular we show that a naive interpretation of exclusion breaks the safety of static typing. We propose some solutions to preserve this safety, either with or without exclusion.

A key idea underlying this article is that there is a thin connection between traits and a subset of AOP. Structural AOP purpose is to localize scattered, related struc-

5. We still need the aliasing operator to get good reusability.
6. A trait containing aliasing also defines a new type but does not break the subtyping relation.

tures in a single entity: so it shares the localization concern with traits. But the reusability concern, although often hidden in AOP, is also of great value – as shown by the AspectJ introduction properties. However traits rely on an explicit mechanism whereas aspects make much more use of implicit properties, thanks to pointcuts. We show here that this process can be made partly explicit.

The structural part of AspectJ is first conceived as a support for behavioral aspects. It lacks some features to deal with conflicts. In particular, the notion of obliviousness in AOP (Filman *et al.*, 2000) is often discussed as it exposes to the same pitfalls that follow from lack of scope in traits model – as shown in (Aldrich, 2004). As underlined in (Douence *et al.*, 2004), interaction detection and composition (that is, conflict resolution) of advices in behavioral AOP is a key feature for aspects. Then the idea of composition and conflict resolution in structural AOP seems relevant: we could pay attention to traits operators in order to improve static programming and weaving. It is noticeable that HyperJ (Ossher *et al.*, 2000), which focuses on structure, proposes some similar operators. Future work involves implementing operators for conflicts resolution in an AspectJ-like extensible weaver: one idea could be to use the Reflex framework (Tanter *et al.*, 2004) extended for structural reflection.

6. References

Aldrich J., « Open Modules: Modular Reasoning in Aspect-Oriented Programming », *Proc. of Foundations of Aspect-Oriented Languages (AOSD'04)*, March, 2004.

Bergel A., Ducasse S., Nierstrasz O., Wuyts R., « Classboxes: Controlling Visibility of Class Extensions », *Computer Languages, Systems and Structures*, vol. 31, n° 3-4, p. 107-126, September, 2005.

Black A. P., Schärli N., Ducasse S., « Applying Traits to the Smalltalk Collection Classes », *in* , R. Crocker, , G. L. Steele, Jr. (eds), *Proc. of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications (OOPSLA 2003)*, ACM Press, Anaheim, California, USA, p. 47-64, October, 2003.

Bracha G., Lindstrom G., « Modularity meets inheritance », *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, IEEE Computer Society, Washington, DC, p. 282-290, 1992.

Douence R., Fradet P., Südholt M., « Composition, Reuse and Interaction Analysis of Stateful Aspects », *Proc. of 3rd International Conference on Aspect-Oriented Software Development (AOSD'04)*, ACM Press, p. 141-150, March, 2004.

Ducasse S., Schaerli N., Nierstrasz O., Wuyts R., Black A., « Traits: A Mechanism for fine-grained Reuse », *Transactions on Programming Languages and Systems*, 2005.

Filman R. E., Friedman D. P., « Aspect-Oriented Programming is Quantification and Obliviousness », *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October, 2000.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W. G., « An overview of AspectJ », *in* , J. L. Knudsen (ed.), *Proc. ECOOP 2001, LNCS 2072*, Springer-Verlag, Berlin, p. 327-353, June, 2001.

Ossher H., Tarr P., « Multi-Dimensional Separation of Concerns and The Hyperspace Approach », *Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer, 2000.

Schärli N., Ducasse S., Nierstrasz O., Black A., « Traits: Composable Units of Behavior », *Proceedings ECOOP 2003*, LNCS, Springer Verlag, July, 2003.

Tanter É., Noyé J., Versatile Kernels for Aspect-Oriented Programming, Research Report n° RR-5275, INRIA, July, 2004.