Evolving a Reflective Language

Lessons Learned from Implementing Traits

Stéphane Ducasse
RMoD, INRIA Lille Nord Europe
LIFL – CNRS UMR 8022
Lille, France

Marcus Denker
PLEIAD Laboratory
DCC University of Chile
Santiago, Chile

Adrian Lienhard
Software Composition Group
University of Bern
Bern, Switzerland

Abstract

Traits are method groups that can be used to compose classes. They do not have a runtime existence and are conceptually folded into the classes that use them. Traits have been implemented in different languages. While implementing them in Smalltalk, our first reflex was to take advantage of the fact that traits are not run-time entities: we optimized the implementation for space and hence shared methods between traits and classes. However, by doing so we broke the introspective API of Smalltalk.

This paper illustrates a more general problem seen in all reflective systems: the implementation serves both as a model for execution and as the model that is exposed to the programmer. There is a conflict of interests between the information necessary for execution and the information the programmer is interested in. In addition, as soon as the implementation is exposed via reflection, we are not free to optimize. As the complete implementation is visible reflectively, there is no way to hide the optimizations.

Few papers report errors and this is one of them. We report our experience facing the initial API mismatch, which has a significant impact on the system because the language is reflective (*i.e.,* written in itself and causally connected). We present the new introspective API we put in place.

Categories and Subject Descriptors D.3.2 [*Programming languages*]: Object-oriented languages; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Languages

Keywords traits, object-oriented languages, smalltalk, reflection, software evolution

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IWST 2009 September 2009, Brest
Copyright © 2009 ACM . . . $10.00

1. Reflection on Language Features

Reflective languages as described by the two following definitions have a causally connected representation of themselves.

> "Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data: providing such an encoding is called reification." [1]

> "A system having itself as application domain and that is causally connected with this domain can be qualified as a reflective system." [11]

A reflective language is a good system to implement new features since it is open and often malleable to changes [6, 9, 12]. However, there is a cost in the fact that a reflective system will reflect on its own implementation. This poses a problem since the implementors are not able to freely optimize the system without taking into account that users can have access via the reflective API to the underlying implementation. [8, 9].

There is a plethora of approaches introducing new features in a language, but rarely do the authors evaluate the cost related to the presence of the underlying language reflective API [2]. In this paper we present the lessons learned from introducing traits and breaking part of the structural MOP of Smalltalk.

Before illustrating the problem, we briefly present traits and also the reflective API of Smalltalk that got affected by the traits presence.

2. Traits in a Nutshell

Since this paper shows how the optimized implementation of traits in Smalltalk broke part of the Smalltalk reflective API, we present traits and stress some particular design points.

Traits are units of behaviour. They are groups of methods that act as behavioural building block of classes [5]. In addition to offering behaviour, traits also *require methods*, *i.e.,* methods that are needed so that trait behaviour is fulfilled. Traits do not define state, instead they require accessor methods.

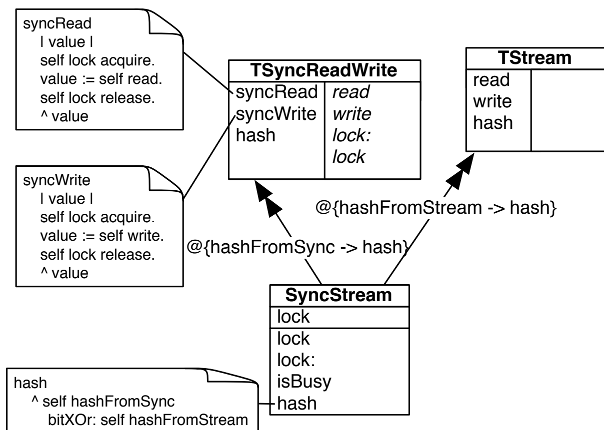

Figure 1. The class SyncStream is composed of the two traits TSyncReadWrite and TStream.

Figure 1 shows a class SyncStream that uses two traits, TSyncReadWrite and TStream. The trait TSyncReadWrite provides the methods syncRead, syncWrite and hash. It requires the methods read and write, and the two accessor methods lock and lock:. We use an extension to UML to represent traits (the right column lists required methods while the left one lists the provided methods).

Explicit composition. A class is then defined by specifying its superclass, instance variables, and used traits. In addition it is the responsibility of the class to explicitly resolve conflicts that may occur when two traits define methods having the same name.

Trait composition is built around the following three rules:

- Methods defined in the composer take precedence over trait methods. This allows the methods defined in a composer to override methods with the same name provided by the used traits.
- In any class composer, the traits can be in principle inlined to give an equivalent class definition that does not use traits.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Conflict resolution. While composing traits, method conflicts may arise. A *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait.

There are two strategies to resolve a conflict: by implementing a method at the level of the class that *overrides* the conflicting methods, or by *excluding* a method from all but one trait. In addition, traits allow method *aliasing*; this makes it possible to introduce an additional name for a method provided by a trait. The new name is used to obtain access to a method that would otherwise be unreachable because it has been overridden [5].

In Figure 1, the class SyncStream is composed from TSyncReadWrite and TStream. The trait composition associated to SyncStream is:

TSyncReadWrite alias *hashFromSync* → *hash*
\+ TStream alias *hashFromStream* → *hash*

The class SyncStream is composed of (i) the trait TSyncReadWrite for which the method hash is aliased to hashFromSync and (ii) the trait TStream for which the method hash is aliased to hashFromStream.

Method composition operators. Trait composition is based on four operators [5]: sum (+), override (▷), exclusion (−) and aliasing (alias →). For further details on trait composition, we refer to the traits paper [5].

Trait methods are then conceptually shared among all the classes using them. As we will see, sharing traits at the implementation level raises some problems in presence of the reflective API.

3. Smalltalk reflective API in a Nutshell

Smalltalk offers a solid structurally reflective API [7, 12] as well as some powerful behavioural capabilities [3, 4]. Here we present only the ones required to illustrate our point.

Class as Objects. The programmer can introspect a class and get access to all its local method selectors as well as the inherited ones.

Point selectors	Returns all the selectors of methods locally defined
Point allSelectors	Returns all method selectors including the superclass ones
Point ≫ #setX:setY:	Returns a compiled method

Compiled methods as meta-objects. While in early Smalltalk implementations a compiled method was an internal object, it slowly acquired a reflective API in later versions. For example, a compiled method knows directly its class in Pharo, while before, all the system classes had to be queried. In addition, a compiled method knows its selector and has a property API to store additional state.

method selector	Returns the selector of the compiled method
method methodClass	Returns the class to which the compiled method belongs

The variable method in the table above holds a compiled method instance. A compiled method can be accessed by sending the message ≫ to a class with the method name as argument, for example, Point ≫ #setX:setY:.

4. Problems

In Squeak Smalltalk, like in most other Smalltalk dialects, a compiled method is installed in the method dictionary of a class. A compiled method is a regular object in memory and serves as a container for the bytecode of a method. The source is stored in external files. As described in Lienhard's Masters thesis [10], the implementation of traits in the Squeak kernel takes the static approach where methods from traits are propagated to classes at *compile time*. (An alternative strategy would have been to modify the method lookup of the VM.)

In this first implementation of traits, we make use of the fact that compiled methods can be shared[1]. When a method is propagated from a trait to another trait or to a class, the very same method object is installed. This means, that method objects are shared between traits and classes. At the time we designed the new traits kernel, the sharing of compiled methods seemed like a reasonable design decision because it avoided to recreate a new compiled method for each class and hence saved memory.

While this optimization was good to optimize memory usage, in the presence of an introspective API it turned out to be a mistake. This became especially obvious when the API of the class CompiledMethod was later extended. For instance, querying for the class in which the method is installed or the method name by which the method can be called, became ambiguous for shared methods. With this design, a correct implementation of this API was not possible because storing the class and name of the method did not make sense for shared methods.

The Case of Aliases. Aliases in traits are not method renames as in Eiffel. An alias is just a new name for an existing method. Aliases are a good solution to let the programmer access a trait method that has been redefined in the class. Without aliases, the programmer would be forced to copy and paste the hidden method. Again with a space optimization in mind, aliases were implemented by adding an entry to method dictionary and by sharing the aliased method. This is a good solution for a non reflective language. The problem is that a compiled method can be asked for its name. It is not clear whether the aliased name or the hidden method name should be returned.

Reflective API Problems. Besides the previous problems that can be solved by not sharing the compiled methods, there is a family of problems due to existing programs and tools not being aware of traits. For example, the code file-outer writes all the methods of a class to disk using the message selectors. Therefore, in presence of traits, the trait methods are saved as well mixed with the local methods. A similar problem occurred at the level of the user interfaces. Browsing a class did not distinguish between method acquired by trait composition and local or inherited ones. These problems were expected since the semantics of the method selectors changed and tools needed to be adapted to the presence of traits.

In summary, the sharing of method objects saved memory but hindered the evolution of a sound reflective API. Therefore, there was a need for a clear structural MOP that we will present in the following section.

[1] One exception are methods with a super message send since they capture the static superclass at which the lookup is started.

5. A Structural reflective API for Traits

The new MOP is based on three parts: the absence of sharing between traits methods, a new API for navigating classes in presence of traits, and a compiled method and trait centered API.

5.1 Class and Traits

Common API of Classes and Traits. Classes and traits have a common interface as both are behaviors holding sets of methods. This means that both traits and classes can often be treated the same. Therefore, even though traits are a new concept, all tools working on classes can be easily adapted to take traits into account.

In the following, we present the common reflective API that is provided by both traits and classes.

traits	A collection of traits that are directly used
allTraits	The transitive closure of all traits used
selectors	The names of all methods installed in the method dictionary (*i.e.*,, methods implemented in classes and traits and methods obtained from traits)
allSelectors	Like selectors, but including superclasses of classes
localSelectors	Like selectors, but excluding methods obtained from traits

Global API to Access Traits and Classes. The global namespace (Smalltalk) contains both classes and traits. The existing API to access classes continues to work with classes:

Smalltalk allClasses	all classes, no traits
Smalltalk allClassesDo: aBlock	iterate over all classes

For traits, the following methods where added:

Smalltalk allTraits	all traits
Smalltalk allClassesAndTraits	all traits and classes
Smalltalk classOrTraitNamed: aString	returns class or trait
Smalltalk allClassesAndTraitsDo: aBlock	iterate over classes and trait

5.2 Non Sharing

The new implementation copies the method instances and their source code on a per use basis (for each class or trait). This way a compiled method has a unique selector and a unique class to which it belongs. The source is also not shared anymore since with aliases the selector of a method can differ from its original (in the old implementation, the source was patched before shown to the user). We propose a finer MOP for compiled method as shown below.

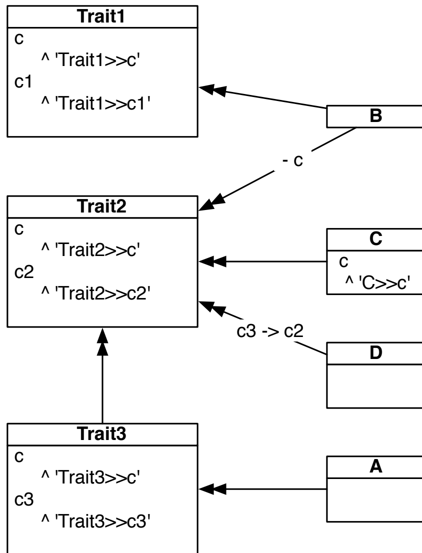

Figure 2. Covering all situations with traits and classes.

As we do not share the methods anymore, there will be a slight increase in instances of class CompiledMethod in the system. We show the increase for Pharo version #10337:

Number of Classes	1803
Number of Traits	75
Number of Methods with sharing	35650
Number of methods with copying	38287
increase	7.39%

5.3 New Compiled Method MOP

While just copying all the trait methods resolves the issues related to the lack of a one to one mapping between a name to access a metaobject and this metaobject, it does not really help clients to build (meta) applications in presence of traits. We present now that just adding one extra concept, the *origin* of a compiled method, results in a solid API. It should be noted that this solution was not that obvious at the beginning and that several iterations were necessary to arrive to this minimal yet expressive API.

Let us take the situation described by Figure 2. While abstract, it illustrates all the cases of trait usage, aliasing and local method definition.

Compiled method methodClass. The message methodClass always returns the class in which the method is either local or used from a trait. methodClass therefore takes the standpoint where all traits are flattened into the classes that use them. If it is sent to a method from a trait, it returns the trait itself.

The following tests illustrate the semantics of methodClass when sent to traits. The traits Trait1 and Trait2 define two methods named c. By default a method defined in a trait answers its trait as its methodClass:

```
self assert: (Trait1>>#c) methodClass = Trait1.
self assert: (Trait2>>#c) methodClass = Trait2.
```

This allows clients to treat traits and classes polymorphically from their method perspective. An alternative design would have been to return nil or raise an exception when sent to a trait.

The method c is redefined locally in the class C. The expression c methodClass returns C since the method is redefined locally. In addition c2 methodClass returns the same result, since the trait Trait2 is used by the class C. In presence of aliases as with the class D, which aliased the method c2 as c3, the methodClass is the class in which the alias occurs.

```
self assert: (C>>#c) methodClass = C.
self assert: (C>>#c2) methodClass = C.

self assert: (D>>#c) methodClass = D.
self assert: (D>>#c2) methodClass = D.
self assert: (D>>#c3) methodClass = D.
```

Finally, even if a trait is composed of other traits (as Trait3 is composed of Trait2 in Figure 2), the methodClass of c2 in class A is A.

```
self assert: (Trait3>>#c) methodClass = Trait3.
self assert: (A>>#c2) methodClass = A.
```

Compiled method selector. The selector of a compiled method is always the selector of its current use. In particular, an aliased method has the name of its alias name.

```
testSelector
  self assert: (A>>#c) selector = #c.
  self assert: (C>>#c) selector = #c.
  self assert: (Trait3>>#c) selector = #c.
  self assert: (Trait3>>#c2) selector = #c2.
  self assert: (D>>#c3) selector = #c3.
```

Compiled method origin. Finally, in contrast to methodClass which always represents the current container of the receiver, we introduced the message origin which returns the defining class or trait of a compiled method. Such message makes the development of tools much simpler since the client does not have to navigate the internal trait composition structure.

The origin of c2 in the trait Trait3 is Trait2 because it is not redefined in Trait3 (same for D≫#c2). Class A gets the method c from the Trait3 and not Trait2 which also defines it, since Trait3 locally redefined it.

```
self assert: (Trait3>>#c2) origin = Trait2.
self assert: (A>>#c2) origin = Trait2.
self assert: (D>>#c2) origin = Trait2.
self assert: (A>>#c) origin = Trait3.
```

Method c is defined locally in class C, which overrides its definition from Trait2. Its origin is then such a class C.

```
self assert: (C>>#c) origin = C.
```

Since class B excluded method c from Trait2 and got the one from trait1. The origin of c is Trait1. Finally the origin of method c3 in class D which is an alias to the method c2 is the class Trait2. Here even if the method C3 is only defined in D the API stresses its origin as a copy of c2. We may change this decision based on user feedback.

```
self assert: (B>>#c) origin = Trait1.
self assert: (D>>#c3) origin = Trait2.
```

6. Discussion

Introducing a reflective API for traits has shown a general problem of reflective systems: as the implementation serves both as the structure for execution and the structure for reflection, any implementation artefact will have an impact on reflection. Any optimization will be visible and can lead to problems like in the case of compiled methods sharing.

In this paper, we solved the problem by not optimizing compiled method sharing. But optimizations like these, especially concerning memory usage or execution performance, can be of great value to make a reflective system practical. Conversely, any definition of a MOP will put constraints on the implementation. Kiczales and Lamping [8] report their problems designing a MOP that would not constrain the implementors.

What we need is the ability to have multiple meta-models being available in the system. A low-level one concerned with execution and one or more higher-level models concerned with providing information for the programmer. By doing so we would be reifying the differences in behavior perceived throughout our analysis. Thus, providing different behavior for the different requirements.

6.1 Mirrors

Bracha and Ungar discuss three principles to build reflective APIs [2]. Encapsulation: meta-level facilities must encapsulate their implementation. Stratification: meta-level facilities must be separated from base-level functionality. Ontological correspondence: the ontology of meta-level facilities should correspond to the ontology of the language they manipulate.

As a solution, Bracha and Ungar propose Mirrors. Reflective capabilities are provided by mirror objects. A solution with mirror-like meta-objects is possible for the problem we encountered in this paper.

We may wonder why a compiled method which is a runtime object needs to know about its class and its selector. This question is valid and interesting. Especially when we see that a compiled method should be accessed via its class using a defined public API (A≫#foo or A compiledMethodFor: #foo).

Let's discuss an alternative design: suppose we introduce a separate compiled method metaobject in addition to compiled methods. In such a design, a compiled method does not act as a metaobject and it only holds information for playing its role; in such a case a compiled method does not know its class nor its selector and as such can be easily shared. Now since we still need a metaobject for compiled method, we introduce a dedicated class CompiledMethodMetaObject, which holds a compiled method, its class and selector. The system should be adapted to return CompiledMethodMetaObjects instead of compiled methods. With such a design, it is interesting to see that the compiled method could be shared but not its metaobject. Now the question of the flow of API use is interesting. Indeed, a compiled method should not hold a pointer to its metaobject as the definition below illustrates. Else we can be in exactly the same situation we describe in the beginning of the paper.

```
CompiledMethod>>selector
  ^ metaObject selector
```

Therefore compiled methods should only be manipulated through their metaobjects. This indicates a layering between the reflective API and its domain. Such a solution goes in the same direction of Mirrors where default Smalltalk reflective objects such as classes are separated into domain objects and their reflective counterpart.

6.2 Towards Multiple Meta-Models

The problem we encountered is actually very interesting for research: how can we put support for multiple meta-models into the reflective core of the language? Mirrors provide a first step towards solving this problem, but we think that more work is needed. For example, it is not clear how to manage the causal connection of the base-level with the multiple meta-models or how to model the connection between multiple meta-models.

7. Conclusion

We have presented a new reflective API concerned with the structure of classes and traits. The realization of this API showed a general problem of reflective systems: the implementation serves as the model for reflection and therefore all implementation-level artefacts are visible for reflection.

We solved the problem by not optimizing for space (not sharing compiled methods) and presented the new reflective API.

The problem presented in this paper is not new. However, while a large body of work introduces new concepts in current reflective languages, really few were concerned about the impact of such introduction and the difficulties faced because of the presence of reflective API.

Beyond the direct scope of the paper, the problems show that more research is needed to make it possible to have multiple meta-models to be able to represent both an optimized model for execution and high-level models for the programmer.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 - Sept. 2010) and "Biologically inspired Languages for Eternal Systems" (SNF Project No. PBBEP2-125605, Apr. 2009 - Mar. 2010).

We thank Jorge Ressia for discussing the concepts presented and his help in reviewing various drafts of this paper.

References

[1] D. Bobrow, R. Gabriel, and J. White. CLOS in context — the shape of the design. In A. Paepcke, editor, *Object-Oriented Programming: the CLOS perspective*, pages 29–61. MIT Press, 1993.

[2] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[3] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[4] S. Ducasse. Evaluating message passing control techniques in Smalltalk. *Journal of Object-Oriented Programming (JOOP)*, 12(6):39–44, June 1999.

[5] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006.

[6] B. Foote and R. E. Johnson. Reflective facilities in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 327–336, Oct. 1989.

[7] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[8] G. Kiczales and J. Lamping. Issues in the design and documentation of class libraries. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 435–451, Oct. 1992.

[9] G. Kiczales and L. Rodriguez. Efficient method dispatch in pcl. In *Proceedings of ACM conference on Lisp and Functional Programming*, pages 99–105, Nice, 1990.

[10] A. Lienhard. Bootstrapping Traits. Master's thesis, University of Bern, Nov. 2004.

[11] P. Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, Dec. 1987.

[12] F. Rivard. Reflective Facilities in Smalltalk. *Revue Informatik/Informatique, revue des organisations suisses d'informatique. Numéro 1 Février 1996*, Feb. 1996.