A new modular implementation for Stateful Traits

Pablo Tesone, Stéphane Ducasse, Guillermo Polito, Luc Fabresse, Noury Bouraqadi

A new modular implementation for Stateful Traits

Pablo Tesone[a,b,*], Stéphane Ducasse[a], Guillermo Polito[c], Luc Fabresse[b], Noury Bouraqadi[b]

[a] *Inria Lille-Nord Europe*
[b] *IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique Automatique, F- 59000 Lille, France*
[c] *Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France*

Abstract

The term traits is overloaded in the literature. In this work we refer to traits as the stateless model and implementation described in Schaerli et al. articles.

Traits provide a flexible way to support multiple inheritance code reuse in the context of a single inheritance language. The Pharo programming language includes the second implementation of stateless traits based on the original version of Schaerli's one. Even if it is the second iteration of such an implementation, it presents several limitations. First, it does not support state in traits. Second, its implementation is monolithic *i.e.,* it is deeply coupled with the rest of the language kernel: it cannot be loaded nor unloaded. Furthermore, trait support impacts all classes, even classes not using traits. In addition, while the development tools include full support to work with classes, trait support is more limited because classes and traits do not present the same Metaobject Protocol (MOP). Finally, being monolithic and integrated in the language kernel, it is difficult to extend this current implementation.

This article describes a new *modular* and *extensible* implementation of traits: it is easily loadable and unloadable as any other package. In addition, classes not using traits are not impacted. Finally, this new implementation includes a new and carefully designed Metaobject Protocol (MOP) that is compatible with both classes and traits. This allows one to reuse existing tools as they do not require special support for traits. Then, following the semantics proposed for *stateful* traits in [BDNW07], we present a new implementation of stateful traits. This

*Corresponding author
Email address: pablo.tesone@inria.fr (Pablo Tesone)

implementation is an extension of our new modular implementation.

We implemented modular traits using specialized metaclasses as our main language extension mechanism. By replacing the implementation we reduced the Pharo Language Kernel size by 15%. This model and implementation are used in production since Pharo7.0 (January 2019).

Keywords: traits, dynamic languages, meta-object protocol, language extension, modular languages, metaclass specialization

---

1. Introduction

The literature contains multiple concepts and implementations named *traits*. Several languages offer traits: Self [HCCU91], Scala [Ode07], SEDEL [BO18], Groovy [KÖ7], Fortress [ACH+05] and Scheme [FFF06]. Language extensions have been also provided for Java [BD16, BDSS13], Javascript [vCM11]. However each of these implementations has its own semantics and considerations. Moreover, these implementations introduce conflicting definitions.

In this paper, we use the term *traits* to refer to traits as in Schaerli's et al. articles [SDNB03, DNS+06, NDS06] and implementation [Lie04].

"A trait is a set of methods, divorced from any class hierarchy. Traits can be composed in arbitrary order. The composite entity (class or trait) has complete control over the composition and can resolve conflicts explicitly, without resorting to linearization." [DNS+06]

In addition, since several variations have been made such as Stateful traits [BDNW07] and freezable traits [DWBN07], we refer to this original model and implementation as *stateless traits*. A variation of such an implementation has been introduced in Perl [RSTT04] and PHP [Loc15]. Moreover, SEDEL [BO18] presents a related toy implementation of traits into a static-typed language that adds first class traits.

Traits provide support for multiple inheritance code reuse in the context of a single inheritance language. They solve the diamond problem [Sak89, Sin94, MA09]. Duplicated methods are extracted as traits which in turn are shared by many classes. Traits are also composable to form other traits [SDNB03, DNS+06, Sch05, NDS06]. Reppy and Turon propose a way of reducing boilerplate code by the use of traits [RT07] without the use of metaprogramming.

The Pharo programming language [BDN+09] includes an implementation of *stateless* traits [Lie04]. It is the reference implementation described in the literature [SDNB03, DNS+06, NDS06]. The Pharo 6.0 implementation of *stateless*

traits fully reimplemented the original version [SDNB03, Lie04]; however this second version has still several limitations at the implementation and model levels.

Implementation level. The implementation is *monolithic*: it is deeply coupled with the rest of the language kernel (Section 2). This prevents Pharo developers to easily load, unload or update the trait support. It restricts the flexibility of the system and makes the bootstrap process more complex [PDF+14]. In particular, classes not using traits have an useless dependency with the Trait implementation (*e.g.,* unused instance variables, conditional code).

Model level. The implemented model presents limitations: Stateless traits do not support state definition in traits. The programmer is then forced to define accessors and initialization in all the classes using a trait that requires this instance state [BDNW07]. *Stateful traits* were proposed to extend traits with state support [BDNW07]. In this alternative, instance variables, accessors and initialization needed by a trait are defined in the trait itself but at the expense of a change of instance layout (using dictionary-based access instead of index access). However, stateful traits were never fully implemented or adopted because of the complexity of modifying the existent monolithic implementation, and updating all the existing tools.

The impact of the introduction of a new concept in an existing model and system is often not taken into account explicitly by language designers. Based on more than 10 years of experience working with traits, this paper reflects on the introduction of traits in an existing system and its impact on existing tools. Doing so, we analyse the design space of the classes and traits API in terms of the Metaobject protocol[1] exposed to the developer.

This paper describes modular implementation for introducing traits in a programming language as well as a new implementation of stateful traits based on

[1] A Metaobject protocol is an API allowing to access and often reflectively change implementation aspects otherwise hidden. Kiczales and al. [KdRB91] defines it as: "First, the basic elements of the programming language - classes, methods and generic functions - are made accessible as objects. Because these objects represent fragments of a program, they are given the special name of Metaobjects. Second, individual decisions about the behavior of the language are encoded in a protocol operating on these Metaobjects - a Metaobject protocol. Third, for each kind of Metaobjects, a default class is created, which lays down the behavior of the default language in the form of methods in the protocol."

the semantics defined in [BDNW07]. The contributions of this work are:

- An analysis of the limitations of the current Pharo *stateless* traits implementation. This analysis goes beyond missing state support (Section 2). It shows that traits should be polymorphic[2] to classes and expose a revisited API to support tool builders.
- A new implementation of *stateless* traits that is compatible with classes by design, providing a smooth integration with existing programming tools (Section 3). This implementation solves the key issues of the compatibility between classes and traits from an API perspective. We redesigned a structural Metaobject Protocol [KdRB91] for classes and traits that takes into account the origin of the methods.
- A new Metaobject Protocol supporting a system with and without traits, both supporting stateless and stateful trais (Section 4).
- A new implementation of stateful traits. We extend our new implementation with stateful traits following the semantics described in the literature [BDNW07]. This new implementation is the foundation of the trait model of Pharo 7.0 and is used in an industrial setup (Section 5).
- A modular implementation. Our new implementation is modular since it is a loadable and extensible library. This new library also ensures that classes not using traits are not impacted because it relies on specialized metaclasses[3] that encapsulate the integration with traits. This modular implementation gives developers the freedom to load or unload this language feature bringing flexibility to the development team [CS17]. It is even possible to modify the library without affecting the whole environment (Section 6).

Compared with the current stateless trait model and implementation in Pharo 6.0[4],

[2] An object exposes the same API than another one with which it is polymorphic. In Java terms, two objects implement the same interface, so they are polymorphic.

[3] A metaclass is the class of a class, thus a metaclass defines how a class behaves. Using a specialized metaclass allows one to modify the properties of classes [KdRB91, DF94, FD99]. Metaclasses provide the flexibility to assign different properties to different classes [BLR98, DSW05, RBY+05].

[4] This paper always refers to version 6.0 of Pharo when talking about the current implementation. And it refers to version 7.0 of Pharo when describing the new implementation.

our solution improves the modularisation of the existing code base: It simplifies it – it reduces the size of the Pharo Language Kernel by 15%. It speeds up the overall bootstrapping process of 30% [PDF+14] for building reduced language kernels [Pol15] (Section 7). Section 8 presents the considerations in the different design decisions, and Section 9 compares our solution with related works. Finally, Section 10 concludes this paper.

2. Limitations of Stateless Traits and its Monolithic Implementation

The stateless traits implementation presents a series of limitations. The analysis of stateless traits limits presented by Bergel et al. in [BDNW07] still applies to the traits implementation in Pharo6.0. We do not repeat it in this article but focus on the practical problems that the Pharo development team faced over the years.

2.1. Vocabulary

In this paper, we use the following common vocabulary: A class defines or has methods and instance variables. Each class has a single superclass it inherits from. A class creates instances. Stateless traits define methods. Stateful traits in addition define instance variable. Traits cannot create instances.

Whenever a class is composed out of a trait, we say that the class *uses* the trait. A class using a trait has access to the methods defined in the trait.

We use the term *traited class* to refer to classes using traits, and *normal classes* to classes without traits. For reading convention, we prefix all traits' names with a T (*e.g.,*TNamed). Finally, the *origin* of a method is the class or the trait that defines it.

2.2. Overloaded Metaobject Protocol

In normal classes, methods originate from two possible locations: Either they are defined in the class itself or they are defined in one of its superclasses *i.e.,* they are inherited. The Metaobject Protocol (MOP) of classes was originally designed to cope with this difference: there exist two families of messages to access the methods [GR89]:

- One family of messages is used to access the methods defined in the class (*i.e.,* selectors, methods).
- The other family, with messages prefixed with "all", is used to access both defined and inherited methods (*i.e.,* allSelectors, allMethods).

This separation of messages provides a clear interface showing where the methods are defined.

To make traits and traited classes compatible with existing tools the previously carefully designed MOP has not been revisited and redesigned, but simply overloaded. For example, the method methods was modified to return not only the methods defined in the class itself but also the ones contributed by the used traits. This is a problem because certain navigation operations such as editing the original method suddenly became a lot more complex to define. To identify method origin, the traited class MOP was modified to provide a new family of methods prefixed with "local" (*i.e.,* localSelectors, localMethods). Similarly to non-prefixed versions of the methods, methods prefixed with "local" return the elements defined in the class itself. However, while the intention was good, the resulting situation is not satisfactory since, having two different MOPs forces tools to use conditional code to detect whether it is manipulating a normal or a traited class to use the correct corresponding API.

2.3. Trait and Class Coupled Implementation

In the current version, the trait implementation is strongly coupled inside the default Pharo metaclass model [BDN+09]. There is no way to have class without trait support. This is a problem because the vast majority of classes do not use traits. In Pharo 6.1, 127 classes use traits over the 6486 classes of the system: it means that less than 1.95% of the classes are using traits[5].

The original objective of having a coupled implementation was to take advantage of trait reuse capabilities. However, the result, while working, makes the code logic more complex. The code that actually integrates traits with classes is embedded inside the standard Pharo metaclasses: Behavior, ClassDescription, Class and Metaclass. Internally, these classes manage both the case where a class uses traits and where they do not, mostly using conditionals. Listing 1 shows an example of conditional code that is scattered all over the language kernel.

[5]Note that this computation does not take into account the fact that when a class root of inheritance uses a trait, its subclasses should be counted as using traits too.

```
Behavior >> isComposedBy: aTrait
  "Answers if this object includes trait aTrait into its composition"
  aTrait isTrait ifFalse: [ ^ false].
  ^ self hasTraitComposition
    ifTrue: [ self includesTrait: aTrait ]
    ifFalse: [ false ]
```

Listing 1: Mixed up implementation in Kernel

This scattered implementation prevents Pharo developers from unloading the trait implementation, or having a smaller kernel [Pol15]. This impacts bootstrap time and future kernel evolution possibilities.

Requirements for the new implementation. Since modularity can be interpreted in different ways, we define precisely what are the requirements for a modular trait implementation. The implementation should be modular in the sense that:

- Classes not using traits should not depend on the trait implementation.
- Trait support should be loadable on demand.
- To support traits, the existing hierarchy of a class should not be modified. To benefit from trait support, a class should not be forced to inherit from a special class. This is particularly important since Pharo only supports single inheritance.

From these requirements we see that the granularity of trait support is a class and its subclasses. It means that using a trait in a class only affects this class and its subclasses but not any of its superclasses.

2.4. Wrong Unified API between Traits and Classes

In the second attempt to propose a better API for traits and classes, changes were made by the community to propose a more unified API to the classes Trait and Class. The initial intent was that a class could understand the same messages than a trait, in particular to avoid tool builders to write complex conditional code. However, this attempt did not fully deliver its objectives. In Pharo6.0 implementation, Trait and Class were both users of the trait TBehavior. The trait TBehavior defined common methods that allowed one to use traits and classes polymorphically in some situations, for example, in the compiler.

However, traits and classes did not provide a fully compatible API. Indeed some class behavior such as instance creation did not make sense on traits. Several

methods in Trait class were cancelled (*i.e.,* throw an error) or had an incompatible implementation. Since traits cannot be instantiated, their method basicNew threw an exception to avoid the creation of instances.

Other meaningless messages contained a valid but arguable implementation (*e.g., superclass* returning nil). This inconsistent and possibly accidental behaviour produced lots of conditional code in existing tools to manage such special cases and avoid misbehaviours.

Listing 2 shows an example of conditional code. This implementation is needed because by design traits return nil to the message instanceSide where classes return the instance side class[6]; and traits by design did not support class variables. That's why, first the developer had to guard against that the argument can be a trait and that the message instanceSide returned nil, then that the argument can be a class trait (a trait applied on the class of a class), to finally be sure that the argument is a class. Listing 2 shows that the situation is even more confusing since it uses cls instead of classOrTrait for the temporary variable name.

As shown, having a non carefully thought API brings more complexity in the methods having to deal with classes and traits.

```
AbstractTool >> browseClassVarRefsOf: aClass
        | cls |
        cls := aClass instanceSide.
        cls isTrait
          ifFalse: [ self systemNavigation browseClassVarRefs: cls ]
```

Listing 2: Conditional code in tools

3. A Modular Stateless Trait Implementation

Implementing a modular and extensible trait implementation presents a number of challenges that we solved in the new implementation presented in this article.

We start with an example of stateless traits as defined by Schärli et al. [SDNB03] (Section 3.1). We remind the reader with the stateless trait composition algebra (Section 3.2). Then, since Pharo's kernel is derived from Smalltalk's one,

[6]In Smalltalk, when a programmer defines a class, the system internally creates a class (e.g., Person) instance of another class that is created automatically: its metaclass called Person class. The message instanceSide makes sure that when sent to a class or its metaclass, the class is returned.

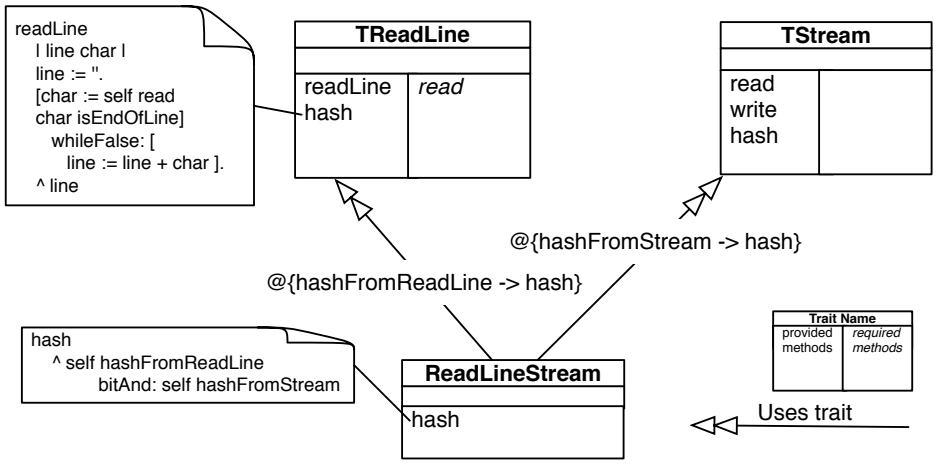


Figure 1: The class ReadLineStream implements a stream that read lines by using two traits TReadLine and TStream.

to ease understanding, we explain and sketch a simplified version of the original Smalltalk kernel that does not include traits [GR89] (Section 3.3). Using the first example, we present our modular stateless trait implementation (Section 3.4).

3.1. Studying a Trait Usage

To make the implementation description clearer, Figure 1 describes a trait-based implementation of a Stream library using the mechanisms described by Schärli et al [SDNB03]. This library gives the ability to read lines. It reads from a stream until an end of line character is encountered.

The class ReadLineStream uses two traits: TStream and TReadLine. The first trait provides the behaviour required to read from the stream and the second one the ability to handle the reading of text lines.

The trait TReadLine defines two methods readLine, and hash, and it requires the method read. We define the TReadLine trait as follows:

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

The trait TReadLine does not use of any other traits. The uses: clause specifies the trait composition (empty in this case).

The interface for defining a class as composition of traits is an extension of the interface used to define classes without using traits. The extended interface includes the uses: clause to express the trait composition. Here we see how ReadLineStream uses the traits TReadLine and TStream:

```
Object subclass: #ReadLineStream
 uses: TReadLine @ {#hashFromReadLine –> #hash }
     + TStream @ {#hashFromStream –> #hash }
 instVarNames: ''
 ....
```

In this example, both traits provide an implementation of hash, producing a conflict as the resulting class cannot have twice the same method. In our example, we decided to have an implementation of hash in ReadLineStream that combine both implementations existing in the traits. This implementation replaces the existing ones in the traits but requires to have access to them. So, we are applying an aliasing operation to the methods contributed by the traits. The hash from TReadLine is renamed as hashFromReadLine and the one from TStream is renamed as hashFromStream. The aliasing of the methods is done through the use of the alias operand (@) in the trait composition of the ReadLineStream class.

3.2. Stateless Trait Composition Algebra

The implemented trait composition follows the semantics defined in the original traits implementation [SDNB03]. The trait composition algebra allows us to compose different traits to form a more complex one. Therefore, classes can use different traits along with a trait composition algebra to express the conflict resolution [DNS+06]. This section can be skipped by a reader knowing trait composition. It is a short presentation of the three traits composition operators that are loaded when the traits library is loaded.

Sequence. The sequence operation allows us to combine two or more existing traits. The traits used in the sequence does not have any priority. The resulting combination has all the methods defined in the original traits. Sequences are created using the + operator.

```
Object subclass: #AnExampleClass
  uses: T1 + T2 + T3
  instanceVariableNames: ''
  ...
```

Alias Method. When two or more traits implements the same method, a conflict occurs. One alternative to solve the conflict is to alias one of the conflicting methods with another name using the @ operator. This operator receives an array of associations (->) of the new name and the original name. The resulting composition has all the methods defined in the trait but with the aliased one renamed.

```
Object subclass: #AnExampleClass
    uses: T1 @ { #new –> #original }
    instanceVariableNames: ''
    ...
```

Remove Method. When a conflict exists between two methods, another alternative to solve it is to remove one of the conflicting methods. The - operator allows us to generate a new composition with a method removed. This composition includes all the original methods but the removed one.

```
Object subclass: #AnExampleClass
    uses: T1 – #aMethodToRemove
    instanceVariableNames: ''
    ...
```

3.3. Starting with a Implicit Metaclass Kernel

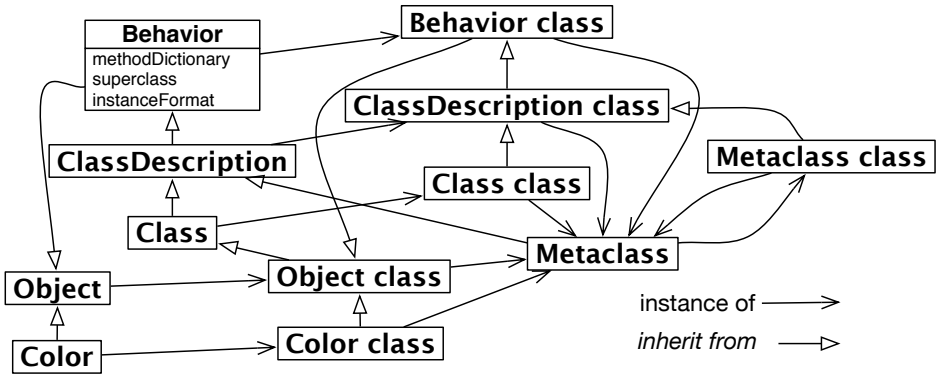


Figure 2: Initial and traditional Pharo kernel (without traits). The class kernel is decomposed in Behavior (root of all metaclasses and classes), ClassDescription, and Class and Metaclass. Behavior defines the essence of a class (superclass, a method dictionary and a way to describe instance state [BDN+09]).

The original Smalltalk kernel that does not include traits [GR89, BLR98, BDN+09]. Figure 2 presents the original metaclass kernel of Pharo and Smalltalk, its ancestor: as in a traditional implicit metaclass kernel [BDN+09]. Each class has its own metaclass *e.g.,* the class Object is instance of the metaclass Object class. The metaclass Object class is instance of the metaclass Metaclass. The metaclass Metaclass is instance of the class Metaclass class and the metaclass Metaclass class is instance of the class Metaclass. Object class inherits from Class.

Figure 3 presents a simplified view of this original kernel. We compacted the Behavior, ClassDescription, Class and Metaclass in the class Class as in CLOS [KdRB91]. Each class in the system is a first class citizen with a superclass, a method dictionary, and an instance format description. The system uses metaclasses to provide a common behaviour to the classes.

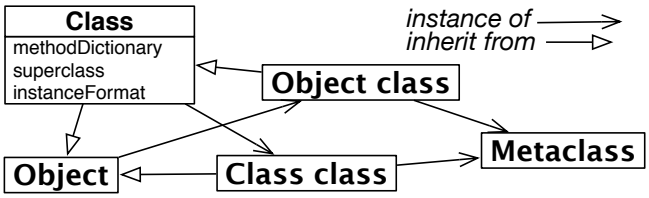


Figure 3: Simplified traditional Pharo kernel (without traits). We compacted the Behavior, ClassDescription, Class and Metaclass in the class Class as in CLOS [KdRB91].

This simplified kernel has no support for traits. Using this simplified kernel we will present the modular implementation of stateful traits that solves the problems mentioned in Section 2.

3.4. Extending the Kernel with a Modular Stateless Trait Implementation

This section presents the extension of the kernel (Figure 3) to support traits as well as an instantiation of this meta-model to represent the trait-based stream library (Figure 1).

To support stateless traits in a modular way, our implementation takes advantage of the existing metaclass support shown in Figure 2. As shown in Figure 4, we define a specialized metaclass named TraitedMetaclass to be used by the traited classes and by the Trait class [KdRB91, BLR98]. With the new implementation:

- Classes using traits are not instances of Metaclass but of TraitedMetaclass.
- Normal classes are kept unmodified, as instances of Metaclass class.

Following the trait model [Lie04], a traited class uses a trait and its traited metaclass uses the corresponding class-side trait. Indeed when a trait defines a method sending a message to its class side, the corresponding method should be available on the composing class [BLR98]. Therefore, traits are defined and applied as a pair of traits (a trait and its class side trait) to a given class.

All the behaviour related to traits is removed from the kernel classes and it is available in the trait implementation. Neither Metaclass, Class or Behavior have

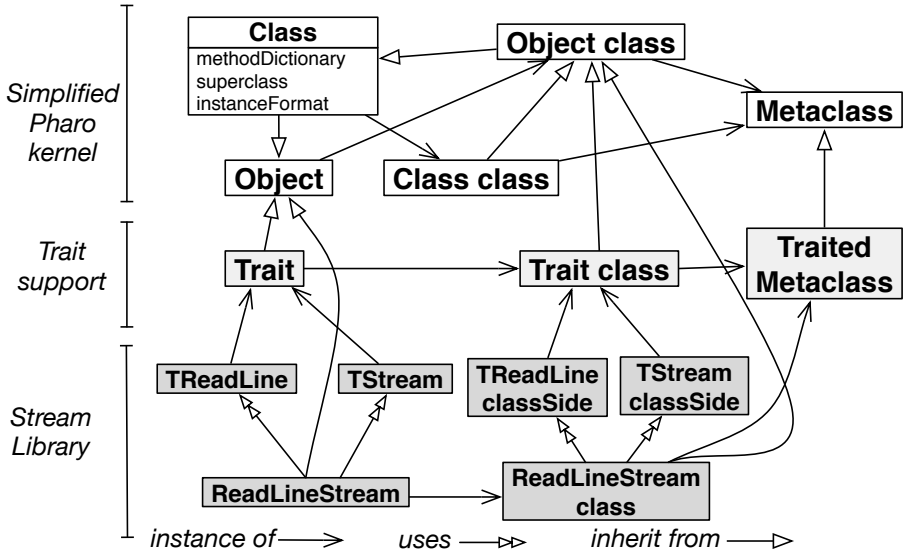


Figure 4: The traits TStream and TReadLine are instances of Trait, their class-side counter parts, TStream classSide and TReadLine classSide are instances of the class Trait class. The class ReadLineStream is an instance of the class ReadLineStream class which is an instance of the new metaclass TraitedMetaclass that manages classes using traits.

any trait related behaviour. As depicted on Figure 4, all the trait behaviour is defined in TraitedMetaclass and in the traited classes. Adding the required behaviour to TraitedMetaclass is a simple operation because TraitedMetaclass is a regular class in the environment, it extends the behaviour of Metaclass through inheritance and overriding. However, integrating the behaviour to support traits into the system is not so simple. We need to integrate this new implementation supporting traits into classes and add new methods and instance variables to handle the trait composition.

In our implementation (presented in this article), the additional behaviour needed in traited classes is factorized in trait named TTraitedClass. This trait is automatically included in all metaclasses of traited classes, as shown in Figure 5. By doing so, the kernel classes of Pharo do not have any reference to the trait implementation and the parallelism of classes and metaclasses hierarchies is preserved.

Using this model, we achieve modularity by packaging Trait, TraitedMetaclass and TTraitedClass in a separate trait library that is imported explicitly by users.

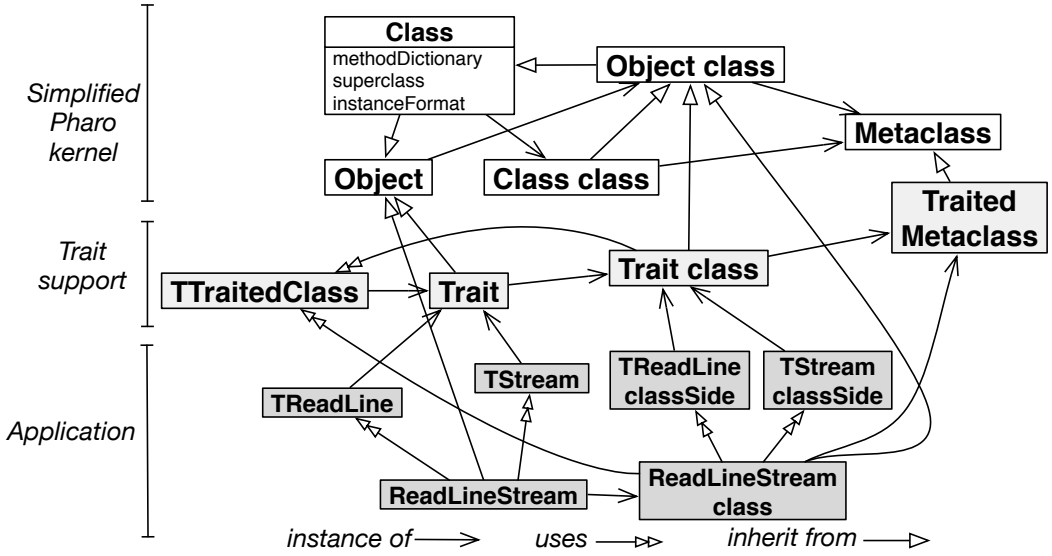


Figure 5: The final model of our implementation, including the TTraitedClass trait. This trait provides the required behaviour to handle traits to the classes using traits.

4. Designing a Modular Metaobject Protocol

To have a good integration between the tools and the underlying model, the Metaobject Protocol has to be clear and consistent. To allow the extension of kernel language with a modular trait implementation, the MOP should also support new extensions in a coherent and compatible fashion.

We propose a MOP with two sets of messages, a set for handling *locally defined elements* and other set to handle *contributed elements*. An element is either a method or an instance variable.

- The *locally defined elements* are the elements directly defined in the class or trait.
- The *contributed elements* are the elements defined elsewhere but nevertheless accessible to the class or trait.

Considering the language kernel without traits, the contributed elements of a class are the set of inherited elements. When the trait library is loaded, the contributed elements of a traited class are the list of both inherited elements and elements defined in the used traits. Note that such contributed element list should

take into account traits semantics: the fact that local elements (from composer class or trait) take precedence over contributed ones (inherited or used traits).

In the example of ReadLineStream (Figure 1), the ReadLineStream class has the locally defined method hash, the contributed methods inherited from Object (for example equals: and printString) and the contributed methods from using the TReadLine and TStream traits.

The Pharo MOP is extended with new messages to support traits. For the sake of clarity, we only present hereafter the messages related to methods. However, the same structure is applicable to other elements of a class such as its instance variables. Note this Metaobject Protocol is shared by classes and traits:

methods. Returns all the methods defined locally in the given class.

selectors. Returns all the method selectors defined locally in the given class.

contributedMethods. Returns all the methods contributed to this class. In the case of normal classes, it includes the methods in the class hierarchy. In the case of traited classes, it includes the methods in the class hierarchy and the methods contributed by the traits.

contributedSelectors. Returns the selectors of those methods returned by contributedMethods.

allMethods. Returns all the methods, including the locally defined methods and the contributed methods.

allSelectors. Returns the selectors of those methods returned by allMethods.

originOfMethod: aMethod. This message returns the class or the trait that defines aMethod.

Using this Metaobject protocol, tools access elements in a class and elements contributed to a class without caring about their origin. The origin of an element is used when modifying the original element (i.e., the method defined in a trait in case of a trait). Moreover, this MOP delegates the resolution of the elements to the corresponding class by exploiting polymorphism to have a modular language extension. The contributed elements of a normal class (without using traits) are calculated by the method defined in Class (Listing 3) and the contributed elements of a traited class are calculated by the method defined in TTraitedClass (Listing 4). So, the implementation in the kernel classes (such as Class) does not know how to

include the contributions from traits, this is resolved in the code implemented in TTraitedClass.

```
Class >> contributedMethods
  "Returns the methods in the superclass
     chain"
  ^ self superclass allMethods
```

Listing 3: Implementation in Class

```
TTraitedClass >> contributedMethods
  "Concatenate the methods from
     superclass and from the used traits"
  ^ self superclass allMethods , self
     traitComposition allMethods
```

Listing 4: Implementation in TTraitedClass

4.1. Traited Class Definition

To have a real modular trait implementation, the syntax to define traited classes is also loaded with the trait library. Listing 5 and 6 show the definition of a traited class (ReadLineStream) and a normal class (Color). As shown, the definition is not the same for traited classes and normal classes, the class definition should be extended to introduce trait composition.

As the definition of classes in Pharo is done through message-sends to the superclass, a new definition is added as an extension method to Class. This extension method allows all the classes in the system to generate new subclasses using traits. A module in Pharo is able to contribute to existing classes through extension methods [PDFT17]. When a module containing extension methods is loaded, the extension methods are installed in the target class.

```
"Defining the class Account
     without traits"
Object subclass: #Color
  instanceVariableNames:
    'red blue green alpha'
  classVariableNames: ''
  package: 'Colors'
```

Listing 5: Class definition without traits

```
"Defining the class Employee using a trait composition"
Object subclass: #ReadLineStream
    uses: TReadLine @ {#hashFromReadLine -> #hash }
        + TStream @ {#hashFromStream -> #hash }
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Streams'
```

Listing 6: Class definition with traits

Each class in the system is responsible to regenerate its own definition. So, each class is asked its definition through the definition message, the new class definition syntax is specialized adequately in the TraitedMetaclass and TTraitedClass.

5. Supporting Stateful Traits

In addition to offer a modular and extensible stateless trait implementation, our discussions with the designers and engineers of the Moose analysis platform show that there was a real need for traits with state. Moose is a software and data analysis platform [NDG05]. Moose meta models represent a lot of data about the language elements and the stateless traits could not capture this important aspect of the representation.

Following the semantics proposed for *stateful* traits in [BDNW07], we extended our new implementation presented in previous sections to support state in traits. This new implementation demonstrates the extensibility of our implementation design by introducing the support of instance and class variables in traits and their initializations.

5.1. Studying a Stateful Trait Example

Before presenting this extension, we first present an example extracted from the original stateful article [BDNW07]. Figure 6 shows this example about an implementation of a SyncStream. This class extends the behaviour of a stream adding a lock to synchronize its accesses.

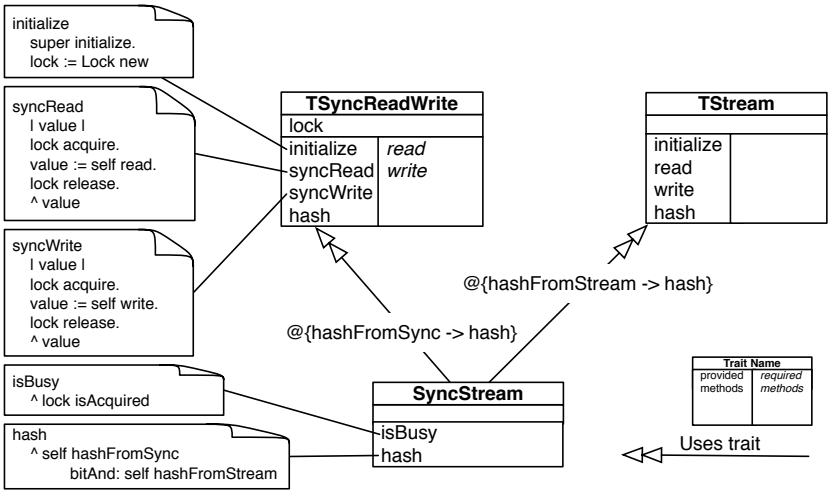


Figure 6: The class SyncStream uses two traits TSyncReadWrite and TStream.

The class SyncStream uses the traits TStream and TSyncReadWrite. The trait TSyncReadWrite defines the variable lock, three methods syncRead, syncWrite and hash, and requires methods read and write.

Following the specifications of the stateful trait model, our new implementation extends the mechanism for defining traits. We define the TSyncReadWrite trait as follows:

```
Trait named: #TSyncReadWrite
    uses: {}
    instVarNames: 'lock'
```

The trait TSyncReadWrite does not use any other traits and it defines an instance variable named lock. The uses: clause specifies the trait composition (empty in this case), and instVarNames: lists the variables defined in the trait (*i.e.,* the variable, lock). The interface for defining a class as a composition of traits is the same as with stateless traits. Here the definition of SyncStream using the traits TSyncReadWrite and TStream:

```
Object subclass: #SyncStream
  uses:  TSyncReadWrite @ {#hashFromSync -> #hash }
    + TStream @ {#hashFromStream -> #hash }
  instVarNames: ''
  ....
```

In our implementation all the instance variables are accessible to methods defined in the class and in all the used traits. In the example, the method isBusy implemented in SyncStream accesses the instance variable defined in TSyncReadWrite. The final set of variables of a traited class contains the variables defined in its hierarchy and the variables defined in the used traits.

When a traited class is defined, its class and instance variables are computed using a simple flattening process. Then methods contributed by its used traits are then re-compiled into this traited class. During this compilation process, the index of the instance variables are recalculated to integrate all the variables defined in the class and contributed by inheritance and used traits.

Note that an implementation based on method copy-down[7] as used in lan-

[7]Copy-down is a way to minimize code repetition in mixin implementation. "Methods that do not access instance variables or super are shared in the mixin. Methods that access instance variables may have to be specialized for the invocation, where the instance variable access is customized according to the structure of instance of the invocation."[BBG+02] The idea is that since the only method impacted by changes to object internal representation (due to mixin application) are methods accessing fields, it is enough to copy down in the subclasses (classes that represent mixin application to another class) the methods accessing fields. All other methods are then applicable to all the mixin application classes.

guages supporting mixins could be evaluated in the future [BBG+02, BC90].

The stateful trait composition algebra extends the one of stateless traits (see Section 3.2). It includes operations required to resolve conflicts arising with instance variables following the semantics described in [BDNW07]. The extended operations are: *renaming* (@@) and *removing* (--) of instance variables.

5.2. Conflict Resolution

When two or more instance variables with the same name exist in the trait composition, there is a conflict. A conflict occurs in two different scenarios: (1) the instance variables have the same name but their values cannot be shared or their usage is incompatible, and (2) the instance variables should be merged as they contain the same information and it is required to be shared by both traits.

```
Trait named: #TNamed
  uses: {}
  instanceVariableNames: 'name'.

Trait named: #TWithRole
  uses: {}
  instanceVariableNames: 'name'.

Object subclass: #Person
        uses: TNamed + (TWithRole @@ {#name -> #roleName})
        instanceVariableNames: ''
        classVariableNames: ''
        package: 'APackage'
```

Listing 7: Example of conflict resolution by renaming the name instance variable of TWithRole into roleName

Listing 7 illustrates the first scenario. The Person class uses two traits (TNamed and TWithRole), both traits define and use the name instance variable. However, these instance variables are used in incompatible ways. The TNamed trait uses it to store the person name, but the TWithRole trait uses it to store the role name. In this scenario the instance variables should be renamed. During the flattening process both instance variables (name and roleName) are created and the methods in TWithRole using the original conflicting variable are rewritten to use the new name (roleName). Renaming is achieved using the @@ operator on TWithRole. This operator creates a new composition with the renamed instance variable definition.

```
Trait named: #TNamed
  uses: {}
  instanceVariableNames: 'name'.

Trait named: #TFormalName
  uses: {}
  instanceVariableNames: 'name prefix'.

Object subclass: #Person
        uses:  TNamed + (TFormalName −− #name)
        instanceVariableNames: ''
        classVariableNames: ''
        package: 'APackage'
```

Listing 8: Example of conflict resolution by merging the two name instances variables coming from TNamed and TFormalName

Listing 8 shows a conflict solved through the merge of two instance variables. In this scenario, both traits (TNamed and TFormalName) are using a name instance variable. Both are using their variable to store the person name, so their usage is compatible. Having a duplicated variable with the same information is not desired. To solve this problem, we use the -- operator to remove the instance variable from TFormalName, leaving the one defined in TNamed. During the flattening process only one instance variable is created and all the methods are using it.

A non-conflicting scenario. This third scenario is about two traits with different instance variables but storing the same information. This is not a conflict, but merging the instance variables is desired to not duplicate the same state. To solve this scenario one of the instance variables is renamed and then the result is merged with the other trait. To merge them, we are removing the name instance variable from the resulting composition.

Listing 9 shows this scenario. The trait TDisplayName stores the person name in the personName instance variable while TNamed stores it in an instance variable named name. As said before, it is not a conflict, but the information should not be duplicated in two different instance variables. To merge, we first rename the instance variable in TDisplayName from personName to name. This will replace all the users of personName with name. Then as both traits have the name instance variable, we remove it from one of the traits. The resulting class will have only one instance variable name.

```
Trait named: #TNamed
  uses: {}
  instanceVariableNames: 'name'.

Trait named: #TDisplayName
  uses: {}
  instanceVariableNames: 'personName'.

Object subclass: #Person
  uses:  TNamed + ((TDisplayName @@ {#personName -> #name}) -- #name)
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'APackage'
```

Listing 9: The instance variables in TNamed and TDisplayName are merged to use the same instance variable.

5.3. Trait Initialization

The initialization of instances of traited classes is an important point to address [BKM12, Nad17]. The question is how to compose and invoke the initialization of used traits from the composite one (either a trait or a class). In particular we do not want that the class initialization has to be changed each time a used trait (of the traits the class uses) changes. We propose a simple strategy to support this.

The initialization of the instance variables defined in traits is performed through the redefinition of the initializeTrait method. Each trait requiring the initialization of instance variables overrides this method. If a trait does not include this method, an empty method is generated during the flattening process. When an object of a traited class is instantiated, the initialization code on the class and the initialization code on the trait is executed. Listing 10 shows how the trait TTraitedClass, which is applied to the class side of all the traited classes, redefines the new method.

A scenario that should be treated correctly is when a traited class uses a trait composition including several traits with initialization code. In this scenario, all the initialization code should be installed in the traited class. To avoid name clash, the different initializeTrait methods are aliased, and a new initializeMethod calling the aliased methods is generated. In Listing 11 the initializeTrait method invokes the initializeTSyncReadWrite and initializeTStream methods that have been generated by the aliasing.

```
TTraitedClass >> new
 ^ self new
    initialize;
    initializeTrait;
    yourself.
```

Listing 10: Implementation of new method.

```
SyncStream >> initializeTrait
 self initializeTSyncReadWrite.
 self initializeTStream.
```

Listing 11: Generated initializeTrait.

5.4. Metaobject protocol support

Our proposed MOP supports stateful traits following the same idea expressed for stateless traits. The same operations described for methods are applicable to instance variables. As said in Section 4, the instance variables related operations have the following semantics:

slots. Returns all the instance variables[8] defined locally in the given class.

contributedSlots. Returns all the instance variables contributed to this class. In the case of normal classes, it includes the instance variables in the class hierarchy. In the case of traited classes, it includes the instance variables in the class hierarchy and the instance variables contributed by the traits.

allSlots. Returns all the instance variables, including the locally defined methods and the contributed methods.

originOfSlot: anInstanceVariable. Returns the class or trait defining the instance variable.

Using this MOP, the tools are implemented without differentiating if they are using normal classes, traits or traited classes.

6. Implementation Details

We validated our solution by replacing the implementation of traits in Pharo 7.0. This implementation is part of the Pharo 7.0 deployed and released version. It includes the implementation described in Section 3, the proposed MOP in Section 4, and the proposed extension in Section 5.

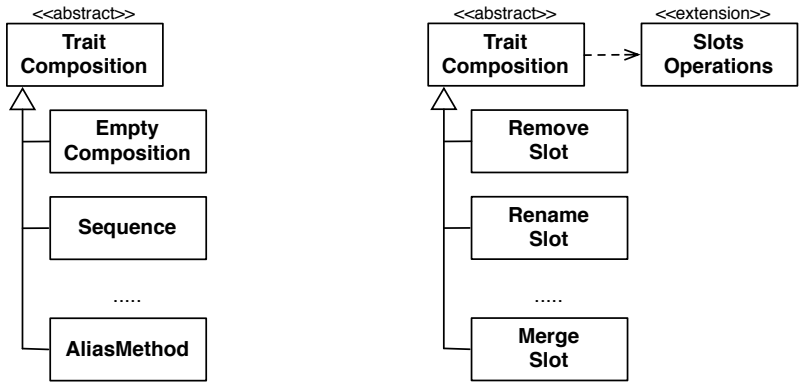
This implementation requires the solution of a number of technical issues. The solutions are presented in this section.

[8] Slots are the new names for first class instance variables in Pharo7.0.

6.1. Polymorphic Classes and Traits

To simplify the integration of traits with existing tools, we represent traits as regular classes. They are subclasses of the Trait class. The only restriction we impose by design is that traits cannot be extended by means of inheritance. The common Trait superclass allows one to easily specialize trait specific methods and implement a clean interface polymorphic with regular classes. For example, the Trait class overrides compile: to compile and install a method locally in a trait and then propagate such a change to all its users.

6.2. Implementing an extensible algebra



(a) Original stateless trait operations.

(b) Stateful trait operation algebra extensions.

Figure 7: Extensible trait composition algebra: the original stateless trait operations are implemented in an extensible way (a) and are extended to support stateful trait operations

To be able to extend the set of operations supported by the system, we implemented the operations of the trait composition algebra as a set of first-class objects. We represent such first-class objects as classes subclasses of TraitComposition. TraitComposition defines methods to handle the composition. We represented first the stateless operations as shown in Fig. 7a. Then such set of operations is extended to support the stateful traits operations and resolution of conflicts (Section 5).

6.3. Flattening and Method Dictionaries

Our solution handles the construction of classes in the same way as the Pharo 6.0's implementation: the methods contributed by a trait are flattened in the class

method dictionary. So, the virtual machine executes the methods defined in the class, the ones contributed by the used traits and the inherited ones using the default VM method lookup, without execution penalties. However, traited classes have two method dictionaries.

- A first method dictionary is used to store all the methods visible to the VM. This method dictionary is internal to the implementation and it is not accessible through the MOP.
- To be fully compatible with the existent tools, traited classes and metaclasses have an additional method dictionary with only the methods defined in the class. The second method dictionary is the one that is accessible through the MOP. This way the outside visible method dictionary has the same semantics than the method dictionary in a normal class. Any change to the available methods of the class is reflected in the hidden method dictionary.

Each method installed in the traited class knows the original trait where it was defined. This information is stored in the additional properties of the method. The access to this information is performed through the class metaobject protocol, and the existence of this additional property is an implementation detail that is again not part of the MOP and as such not exposed to the user.

In the Pharo implementation we see that the memory impact of having a second method dictionary is minimal. This second method dictionary shares the symbols and methods with the default one. The implementation only requires extra memory for the collection itself. Measuring the Pharo 7.0 image the average number of local methods in the 545 classes using traits is 2. So, the average impact per class is of 32 bytes in 32 bits images, and 64 bytes in a 64 bits image (4 instance variables for the dictionary itself and 2 instance variables for each of the 2 associations used).

6.4. Supporting Live Programming

Pharo is a live programming environment. It allows the developer to modify its code while executing it. So, any change made to a trait should be propagated to the users of that trait. To do so, a trait knows all its users and when it is modified (*i.e.,* method or instance variable modification) it then notifies its users. Also the users of a trait are subject to change (*i.e.,* a class is defined to use the trait, or a class using the trait is modified not to use it any more).

This propagation mechanism is implemented using two strategies. First, the notification mechanism is implemented through an *announcement* event system. The announcement event system generates events whenever the system is modified. All the tools requiring to react to these changes are subscribed to these events. Second, a trait intercepts its modifications by overriding the corresponding methods in the Trait class.

The announcement event system is used to update the users of a trait when a traited class is created, modified or removed from the system. The modification overriding is used when a trait is modified (*i.e.,* methods or instance variables) and the users should be updated.

6.5. Class Builder and Installer Replacement

The creation and installation of classes in Pharo are performed by a class Builder and a class Installer. These system components perform all the operations needed to create, modify and install the classes in the environment. To be able to have a modular implementation of traits, some changes to them were needed.

Our solution replaces the class builder and installer with variants that support extensions. It implements a new set of class builder and installer, called the ShiftClassBuilder and ShiftClassInstaller. By default, this new implementation does not have support for traits, as it is intended to be included in Pharo kernel without traits. However, this class builder allows the extension of the building process via plugin, called enhancers.

The class builder and installer are configurable using different builder enhancers [TPF+18]. The builder enhancers contribute to the building and installation process (Figure 8). It also allows one to configure the class of the metaclass to use and how it is created. The implementation of traits is managed by a specific enhancer: TraitBuildEnhancer. It configures the class and metaclass to have the instance variables added by the traits, it also installs all the methods provided by the trait composition. This is performed through the overriding of the defaults actions of the build enhancer.

7. Validation

As said before, the solution has been applied to Pharo 7.0. This allows us to minimize the language kernel that needs to be bootstrapped in the creation of a

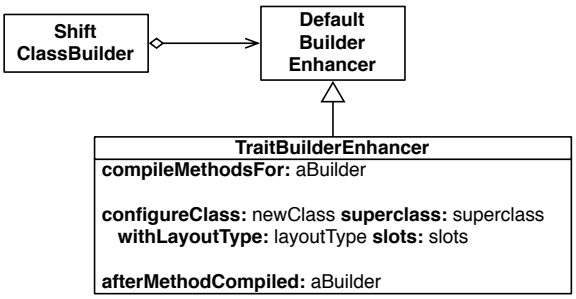


Figure 8: The ShiftClassBuilder, used by our implementation, is extensible through pluggable build enhancers

Pharo 7.0 image[9]. The Pharo 7.0 image is created after each commit from scratch, generating a Pharo image from source code [PDF+14]. In the image generation process, a small image is bootstrapped. This image contains the minimal language kernel to be able to load code. After this minimal image is created all the libraries and tools are loaded, including the new trait implementation. This image is the one that is shipped.

Removing Traits from Kernel. One of the objectives implementing trait support as a library was to remove the trait support from the Pharo language kernel. The language kernel includes all the elements needed to load new pieces of code and to execute Pharo code. The removal represents a diminution of 22,606 lines of code (15.36%) and 2,897 methods (20.79%)[10]. Table 1 shows a detail of removed code. Having a smaller kernel allows us to create smaller images with only the required functionality.

Removing the trait implementation from the language kernel speeds up the bootstrap process of 30%, going from an initial bootstrap process of 16 minutes to a bootstrap process of 11 minutes. On the contrary, the time to load the libraries has increased of only 1 minute. Producing an overall speed up of 20% for the whole process[11].

[9] A Pharo image is a platform agnostic binary file containing all the objects representing the system: classes of the complete Pharo distribution. A default image consists in around 6000 classes and 120,000 methods.

[10] This information was extracted from the Pharo Github repository calculating the difference before and after applying the changes.

[11] Information recovered from the Pharo CI Server Statistics
https://ci.inria.fr/pharo-ci-jenkins2/job/TestpendingpullrequestandbranchPipeline/job/

	Old Impl.	New Impl	Diminution
Lines	147,248	124,642	22,606 (15.35%)
Packages	49	44	5 (10.20%)
Classes	694	587	107 (15.42%)
Methods	13,937	11,040	2,897 (20.79%)

Table 1: Reduction of code after removing traits from Pharo Language Kernel

About traits and classes polymorphism. The previous implementation tried to define a polymorphic API for classes and traits. It leads to ad-hoc situations where while both entities offered the same API, the developers had still to be aware that he manipulated a class or a trait (for example, it does not make sense to display the superclasses of a trait). It produced unnecessary conditional code to differentiate traits and classes. This problem is not only present in the kernel, but also in all the tools and libraries manipulating traits and classes of the new implementation.

Some of the tools affected are the compiler (*Opal*), version managers (*Epicea* and *Monticello*), code navigation (*GT-Spotter*), modeling framework (*Ring*) and class browser and editor (*Nautilus* and *Calypso*). The new implementing represents a reduction of 6,557 lines of code distributed in 89 packages. This impact is visible in the language kernel, but it is more important when analyzing the full system.

Moose. Moose is a software and data analysis platform [NDG05]. Since its version 7.0 it uses stateful traits as a key basic brick for modeling languages (Java, Pharo, PostgreSQL, PowerBuilder, Ada, VisualBasic...) and performing software evolution analyses. The core of Moose defines 123 stateful traits representing elementary program concerns. Such traits are then reused to represent different language and language constructs. This is not a real validation but Moose latest version is a heavy user of stateful traits.

8. Discussion

We discuss now the impact of the new implementation and revisit some design decisions.

development/

8.1. Tool Support

Modern IDEs are composed of many dedicated tools. Pharo offers out of the box: message browsers, a cross referencer, navigation tools, package/class browser, change browser, advanced object inspectors, debuggers, automatic completion, VCS support, and static quality rule engines to name a few.

A loadable trait implementation should include all the resources and modifications needed by the tools to fully support them. Due to the fact that our traits are polymorphic to classes, we are able to remove most of the conditional code in the tools. Some tools should be adapted to use the new Metaobject Protocol for normal and traited classes. However, this topic is still an ongoing effort, as a definitive integration requires modifying existing tools and most of the tools are not prepared to be extended or to allow inclusion of pluggable components. The same consideration is applicable to the new class definition needed by traits. There exists tools that parse the class definition syntax, expecting to receive a given format. These tools should be modified to allow an extensible syntax or to use the messages exposed in the MOP of the classes.

The work left for a definitive integration in Pharo is out of the scope of this work and it will take a couple of years. It is related to the overall quality improvement of existing tools. Pharo is a complex platform composed of about 400 packages and aggregating several subprojects. As such modifying all the required elements is a continuous process requiring time, since such process should always provide useful versions to the users of the platform. This work will be handled with the Pharo community and its industrial consortium.

8.2. About design

In our solution we decided to separate the implementation of traits from the kernel classes through the use of a different metaclass for traited classes. This way all the trait related behaviour is loaded as a library. Other possible alternative is having an extensible metaclass in the kernel of the language. However, making an extensible metaclass in the kernel requires having a more complex kernel than having a simpler implementation.

As explained when presenting our solution (Section 3.4), we have decided to define the behaviour requiring to implement traits in a trait, named TTraitedClass, that is included whenever a traited class is created. The TTraitedClass trait includes all the methods that should be overridden in the Class class to support traited classes. By doing so, we allow one to have the ability to inherit from any class in the system and this without modifying the Class class.

Our solution extends the existing MOP with messages to calculate the origin of class elements. This decision makes the tools independent of the reuse mechanism used. If a given tool wants to modify the definition of an element, the defining class is easily reachable and modifiable.

We decided to have the definition of traits as classes. Another alternative is the definition of traits as independent objects. As it is desirable to have polymorphism between traits and classes to share tool support, they should implement the same MOP. Taking advantage of traits being classes allows a simple implementation of the shared Metaobject Protocol.

We have decided to have two different method dictionaries. By doing so, the VM uses the method dictionary with the flattened traits, and the tools only see the methods through the MOP. This decision creates a separation between the runtime model expected by the VM and the logical model exposed by the tools. Also by restricting the tools to use the method dictionary with the local defined method only, allows us to reuse the implementation of all the methods in Class class; as these methods expect the method dictionaries to only have local defined methods.

Our solution requires a modified version of the class builder and installer. This version allows the extension of the class building process.

9. Related Work

Other languages, such as Scala [Ode07], Rust [MKI14], Groovy [KÖ7] or Self [HCCU91] implement traits as a part of their language kernel. They are integrated in the language and cannot be unloaded. In addition traits in Scala, Rust and Self do not support the same composition semantics than traits defined in [SDNB03, DNS+06]. Their implemented semantic is closer to Mixins [BC90].

There are languages, such as Javascript [Fla97, vCM11], Python [Lut01] and Racket[12], that also implement traits in a loadable library. However, these implementations have been developed for a language kernel without trait support. So, there is no need to modify the existing language kernel or the given tools. Finally, the tools for these languages do not support traits.

There are solutions to add traits to existing static-typed languages [BO18, FR04] and specially to Java [BD16, BDSS13, NDRS05]. However, these solutions focus on the types and type checking challenges introduced by traits. They modify the compiler or generate equivalent code as the one using traits. They do not provide tools or extensible support for development using traits.

[12] http://racket-lang.org/

Pharo already includes a working traits implementation [DNS+06], although this implementation is heavily coupled with the kernel of the language, making impossible to make it a loadable library [Lie04].

Talents are instance specific traits. The early prototype implementation [RBN12] provides ways of expanding Pharo through a loadable library, but this solution has a bigger impact in the performance as the lookup of methods is solved in the image without taking advantage of the performance optimizations of the VM. In addition, talent methods are defined as plain strings passed as arguments to methods and there is no tool support of any kind.

Razavi et al. [RBY+05] already extended an existing language through the definition of custom metaclasses. However, they applied such solution to implement Adaptive Object Models. They do not apply the solution to modify the reuse mechanisms available in the language.

Cazzola et al. [CS17] also provided support to modify the semantic of the language and the reuse mechanisms through dynamic loadable modules. However, their solution is centred on a modular interpreter architecture. They modify the language modifying the interpreter of it. This solution is not applicable for performance reasons.

Bouraqadi et al. [BLR98, Bou04] show how classes are extended in a reflective language using metaclasses. Metaclasses are then used as an extension mechanism to add new properties to class in a per-class basis. We leverage also metaclasses to implement traits as they did for example with Mixins. Differently from them, we also analyse the impact of changing the current trait implementation in the programming environment and its provided tools. We also introduce a modular reflective MOP that prevents new language extensions to modify the existing runtime keeping compatibility.

Malayeri et al. presents CZ [MA09]. CZ is an alternative to traits to solve the diamond problem. It uses an alternative way of combining classes extending the multiple inheritance with restrictions and imports. However, this solution is centred in the development of a language and does not provide tools or a runtime library.

Tesone et al. [TPF+18] shows a general architecture to implement modular extensions to a programming language preserving the performance while running in a single inheritance VM. It shows how a modular design of the underlying mechanisms (i.e., class builder) supports the introduction of mixins, traits, and multiple inheritance. The implementation presented in the present article uses the same idea (i.e., metaclasses) and a modular class building process. However, it is less general and focused on traits. In addition, it solves the more general problem

of class and trait API mismatch and designs an adequate meta-object protocol taking traits into account.

10. Conclusion

In this paper, we addressed the problem of how to modularize a core part of Pharo environment. We took the existing implementation of traits and separated it from the language kernel. Once we had all the mechanisms identified, we proposed a novel implementation that uses a specialization of kernel metaclasses to support traits. This new implementation exposes a common MOP with the existing classes. Since this implementation is based on the class implementation mechanisms, it offers a better support for tools support.

The use of our proposed solution and its implementation allowed us to reduce the size and complexity of Pharo Language Kernel. This reduction speeded up the bootstrapping process in 30%. A faster bootstrapping process improves the ability to modify the language, the infrastructure and the tools. This result is crucial for having a community based programming environment that is fully tested and constantly built from sources. Results of this impact are daily noticeably in the results of the Pharo Continuous-Integration process[13].

Finally, having a loadable independent stateful traits library also achieves other two goals: (1) it produces an increase of the modularity of the system allowing users to select the features used by their applications, and (2) opens the door to the research and implementation of alternative reuse models using similar dynamic and late-bounding techniques. The first objective allows users of Pharo to easily produce tailored configurations for their development and final applications.

Acknowledgements

The following works is supported by I-Site ERC-Generator Multi project 2018-2022. We gratefully acknowledge the financial support of the Métropole Européenne de Lille.

References

[ACH+05] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L Steele Jr, Sam Tobin-Hochstadt,

[13] https://ci.inria.fr/pharo-ci-jenkins2/ and https://github.com/pharo-project/pharo/

Joao Dias, Carl Eastlund, et al. The fortress language specification. *Sun Microsystems*, 139(140):116, 2005.

[BBG+02] Lars Bak, Gilad Bracha, Steffen Grarup, Robert Griesemer, David Griswold, and Urs Hölzle. Mixins in Strongtalk. In *ECOOP '02 Workshop on Inheritance*, June 2002.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, October 1990.

[BD16] Lorenzo Bettini and Ferruccio Damiani. Xtraitj: Traits for the java platform. *Journal of Systems and Software*, 131, 07 2016.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BDNW07] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful traits. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC'06)*, volume 4406 of *LNCS*, pages 66–90. Springer, August 2007.

[BDSS13] Lorenzo Bettini, Ferruccio Damiani, Ina Schaefer, and Fabio Strocco. Traitrecordj: A programming language with traits and records. *Sci. Comput. Program.*, 78(5):521–541, May 2013.

[BKM12] Viviana Bono, Jarek Kuśmierek, and Mauro Mulatero. Magda: a new language for modularity. In *ECOOP 2012–Object-Oriented Programming*, pages 560–588. Springer, 2012.

[BLR98] Noury Bouraqadi, Thomas Ledoux, and Fred Rivard. Safe metaclass programming. In *Proceedings OOPSLA '98*, pages 84–96, 1998.

[BO18] Xuan Bi and Bruno C d S Oliveira. Typed first-class traits. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[Bou04] Noury Bouraqadi. Safe metaclass composition using mixin-based inheritance. *Journal of Computer Languages, Systems and Structures*, 30(1-2):49–61, April 2004.

[CS17] Walter Cazzola and Albert Shaqiri. Open programming language interpreters. *The Art, Science, and Engineering of Programming*, Vol. 1, 2017.

[DF94] Scott Danforth and Ira R. Forman. Derived metaclass in SOM. In *Proceedings of TOOLS EUROPE '94*, pages 63–73, 1994.

[DNS+06] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, March 2006.

[DSW05] Stéphane Ducasse, Nathanael Schärli, and Roel Wuyts. Uniform and safe metaclass composition. *Journal of Computer Languages, Systems and Structures*, 31(3-4):143–164, December 2005.

[DWBN07] Stéphane Ducasse, Roel Wuyts, Alexandre Bergel, and Oscar Nierstrasz. User-changeable visibility: Resolving unanticipated name clashes in traits. In *Proceedings of 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 171–190, New York, NY, USA, October 2007. ACM Press.

[FD99] Ira R. Forman and Scott Danforth. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming*. Addison-Wesley, 1999.

[FFF06] Matthew Flatt, Robert Bruce Findler, and Matthias Felleisen. Scheme with classes, mixins, and traits. In Naoki Kobayashi, editor, *Programming Languages and Systems*, pages 270–289, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Fla97] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, second edition, January 1997.

[FR04] Kathleen Fisher and John Reppy. A typed calculus of traits. In *Proceedings of the 11th Workshop on Foundations of Object-oriented Programming*, 2004.

[GR89] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[HCCU91] Urs Hölzle, Bay-Wei Chang, Craig Chambers, and David Ungar. *The SELF Manual*. Computer Systems Laboratory of Stanfor University, 1991.

[KÖ7] Dierk König. *Groovy in action*. Manning, 2007.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Lie04] Adrian Lienhard. Bootstrapping Traits. Master's thesis, University of Bern, November 2004.

[Loc15] Josh Lockhart. *Modern PHP: New features and good practices*. O'Reilly Media, Inc, 2015.

[Lut01] Mark Lutz. *Programming Python (2nd edition)*. O'Reilly & Associates, Inc., 2001.

[MA09] Donna Malayeri and Jonathan Aldrich. Cz: multiple inheritance without diamonds. In *ACM SIGPLAN Notices*, volume 44, pages 21–40. ACM, 2009.

[MKI14] Nicholas D Matsakis and Felix S Klock II. The rust language. In *ACM SIGAda Ada Letters*, volume 34, pages 103–104. ACM, 2014.

[Nad17] Marco Naddeo. *A Modular Approach to Object Initialization for Pharo*. Theses, Dipartimento di Informatica, Università degli Studi di Torino ; Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille, November 2017.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the European Software Engineering Conference*, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[NDRS05] Oscar Marius Nierstrasz, Stéphane Ducasse, Stefan Reichhart, and Nathanael Schärli. Adding traits to (statically typed) languages. 2005.

[NDS06] Oscar Nierstrasz, Stéphane Ducasse, and Nathanael Schärli. Flattening Traits. *Journal of Object Technology*, 5(4):129–148, May 2006.

[Ode07] Martin Odersky. Scala language secification v. 2.4. Technical report, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, March 2007.

[PDF+14] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Benjamin van Ryseghem. Bootstrapping reflective systems: The case of pharo. *Science of Computer Programming*, 2014.

[PDFT17] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, and Camille Teruel. Scoped extension methods in dynamically-typed languages. *The Art, Science, and Engineering of Programming*, 2(1), August 2017.

[Pol15] Guillermo Polito. *Virtualization Support for Application Runtime Specialization and Extension*. PhD thesis, University Lille 1 - Sciences et Technologies - France, apr 2015.

[RBN12] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-centric debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012.

[RBY+05] Reza Razavi, Noury Bouraqadi, Joseph Yoder, Jean-François Perrot, and Ralph Johnson. Language support for adaptive object-models using metaclasses. *Computer Languages, Systems & Structures*, 31(3):199–218, 2005.

[RSTT04] Allison Randal, Dan Sugalski, Leopold Totsch, and Leopold Tötsch. *Perl 6 and Parrot Essentials*. " O'Reilly Media, Inc.", 2004.

[RT07] John Reppy and Aaron Turon. Metaprogramming with traits. In *European Conference on Object-Oriented Programming*, pages 373–398. Springer, 2007.

[Sak89] Markku Sakkinen. Disciplined inheritance. In S. Cook, editor, *Proceedings ECOOP '89*, pages 39–56, Nottingham, July 1989. Cambridge University Press.

[Sch05] Michel Schinz. Compiling scala for the java virtual machine. *EPFL,*, 2005.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[Sin94] Ghan Bir Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 5(1):34–43, January 1994.

[TPF+18] Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. Implementing Modular Class-based Reuse Mechanisms on Top of a Single Inheritance VM. In *SAC 2018: SAC 2018: Symposium on Applied Computing*, Pau, France, April 2018.

[vCM11] Tom van Cutsem and Mark Miller. traits.js – robust object composition and high-integrity objects for ecmascript 5. In *Proceedings of PLASTIC'11*, 2011.