Implementing Modular Class-based Reuse Mechanisms on Top of a Single Inheritance VM

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

Implementing Modular Class-based Reuse Mechanisms on Top of a Single Inheritance VM

Pablo Tesone
Inria Lille-Nord Europe
IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique Automatique, F- 9000 Lille, France
pablo-adrian.tesone@imt-lille-douai.fr

Guillermo Polito
Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France
guillermo.polito@inria.fr

Luc Fabresse
IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique Automatique, F- 59000 Lille, France
luc.fabresse@imt-lille-douai.fr

Noury Bouraqadi
IMT Lille Douai, Univ. Lille, Unité de Recherche Informatique Automatique, F- 59000 Lille, France
noury.bouraqadi@imt-lille-douai.fr

Stéphane Ducasse
Inria Lille-Nord Europe, France
stephane.ducasse@inria.fr

ABSTRACT

Code reuse is a good strategy to avoid code duplication and speed up software development. Existing object-oriented programming languages propose different ways of combining existing and new code such as *e.g.,* single inheritance, multiple inheritance, Traits or Mixins. All these mechanisms present advantages and disadvantages and there are situations that require the use of one over the other. To avoid the complexity of implementing a virtual machine (VM), many of these mechanisms are often implemented on top of an existing high-performance VM, originally meant to run a single inheritance object-oriented language. These implementations require thus a mapping between the programming model they propose and the execution model provided by the VM. Moreover, reuse mechanisms are not usually composable, nor it is easy to implement new ones for a given language.

In this paper, we propose a modular meta-level runtime architecture to implement and combine different code reuse mechanisms. This architecture supports dynamic combination of several mechanisms without affecting runtime performance in a single inheritance object-oriented VM. It includes moreover a reflective Meta-Object Protocol to query and modify classes using the programming logical model instead of the underlying low-level runtime model. Thanks to this architecture, we implemented Stateful Traits, Mixins, CLOS multiple inheritance, CLOS Standard Method Combinations and Beta prefixing in a modular and composable way.

SAC 2018, April 9–13, 2018, Pau, France

ACM ISBN 978-1-4503-5191-1/18/04…$15.00
https://doi.org/10.1145/3167132.3167244

CCS CONCEPTS

• **Software and its engineering → Source code generation**; **Runtime environments**;

KEYWORDS

code reuse, composable language features, performance

ACM Reference format:
Pablo Tesone, Guillermo Polito, Luc Fabresse, Noury Bouraqadi, and Stéphane Ducasse. 2018. Implementing Modular Class-based Reuse Mechanisms on Top of a Single Inheritance VM. In *Proceedings of SAC 2018: Symposium on Applied Computing , Pau, France, April 9–13, 2018 (SAC 2018),* 8 pages.
https://doi.org/10.1145/3167132.3167244

1 INTRODUCTION

Existing object-oriented programming languages propose different ways of reusing and combining existing and new code, *e.g.,* single inheritance [15], multiple inheritance [18], Traits [17] or Mixins [11]. All of these mechanisms present advantages and disadvantages. Ideally, depending on the characteristics of the software under development, the software developer should be able to decide which mechanism to use, although in reality this is not true for most scenarios.

Each programming language implements only a subset of these mechanisms. When designing a language, the designer selects the set of reuse mechanisms that best fits the desired characteristics of the language. For example, Java only implements single inheritance, Pharo implements single inheritance and stateless traits, Scala [31] implements single inheritance and Mixins[1], CLOS [16] implements multiple inheritance and *after*, *around* and *before* methods.

Several programming languages (*e.g.,* Pharo [7], Groovy [26], Scala, Jython [24]) are implemented on top of well-known high performance virtual machines (VMs) to benefit from the existing infrastructure and optimisations such as the HotSpot Java Virtual Machine [28], the .NET Common Language Runtime [10], the Android Runtime [20] and the Cog Smalltalk Virtual Machine [29].

[1]It also implements the Trait concept but only as a variation of mixins. It is intended to prevent the diamond problem [31].

However, these VMs are originally only meant to execute single inheritance object-oriented languages. Indeed, they implement special optimisation techniques such as stack-to-register mapping [23], polymorphic inline caches [22], just-in-time compilation [14] and adaptative optimizations [2] assuming a single-inheritance object-oriented language. To support their reuse mechanisms on top of these VMs, language designers have to map the programming language model to a runtime model as the one expected by the VM.

Therefore, implementing different reuse mechanisms on top of a single inheritance VM works at two levels, the *logical level* and the *VM runtime level* [33]. In the logical level a program is represented using the provided language constructs. In the VM runtime level the program is represented as low level structures representing classes that use single-inheritance. The program representation in the logical level is used to generate the classes in the VM runtime level. When the VM runtime level is executed the result is the same as the expected execution of the logical level [33]; both models have the same semantics when executed.

Moreover, current programming languages do not allow the easily combination or definition of new code reuse mechanisms. The set of available mechanisms are embedded in the language implementation. The only way a developer is able to select the set of reuse mechanisms is by selecting a different programming language. However, this is not always possible. For example, if older parts of the application are already implemented with a given language changing the programming language requires rewriting the existing code. In addition, a developer cannot choose to use different reuse mechanisms in two different parts of her program *e.g.,* using Traits for some classes and multiple inheritance for other classes.

The main contribution of this paper is a modular meta-level runtime architecture that: (1) supports different reuse mechanisms that can be applied to different parts of an application; (2) supports the combination of these reuse mechanisms; (3) provides a clear meta-object-protocol (MOP) to implement new reuse mechanisms that can be efficiently executed on a single inheritance VM; and (4) provides a reflective MOP to access and modify the logical model. Propagating the changes to the runtime model. This MOP allows the polymorphic use of different reuse mechanism during runtime. Finally, we have validated our solution by implementing a prototype in Pharo.

2 A META-LEVEL ARCHITECTURE TO MODULARIZE REUSE MECHANISMS

We propose a new meta-level architecture that transforms a program *logical model* into the *runtime model* expected by the VM (Figure 1). This meta-level architecture allows us to transform a logical model adding different language features, for example using different reuse mechanisms. A program is defined at the base-level, in terms of a logical model constituted by logical classes. A logical class is then extended by using a *contributing part*. The contributing part exposes a meta-object protocol (MOP) that language designers extend to define new reuse mechanisms such as traits or multiple inheritance. A new language feature is implemented in the meta-level through the extension of ContributingPart class. Once a new contributing part is available in the meta-level, it is possible to use it in logical classes.

A *class builder* takes as input a *logical class* and creates a *runtime class* from it in collaboration with the contributing part. This *runtime class* is a class as expected by the VM *i.e.,* it contains the class methods, instance variable definitions and superclass, as required by a single-inheritance model. This approach is similar to the one of multiple inheritance or mixin implementation based on inheritance linearization [11]. The VM executes the code in the generated runtime classes, taking advantage of all its high performance optimizations.

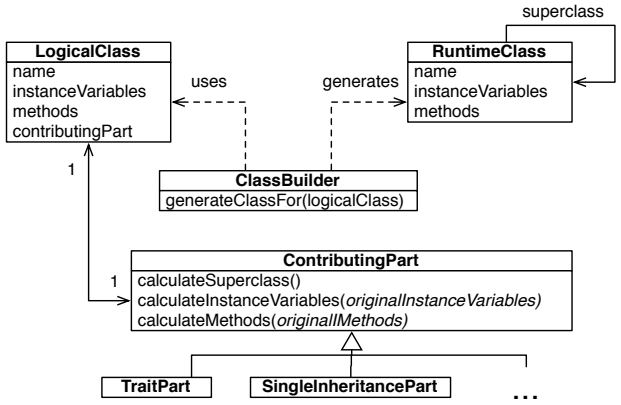

Figure 1: Meta-Level Architecture Components. Logical classes have a contributing part defining what reuse mechanism it uses. A class builder uses a logical class to generate a runtime class following the single inheritance model as the VM requires.

2.1 Automatic Accessor Generation: Our Architecture By Example

Let's imagine we want to extend our language with automatic accessor (getter and setter) generation. We chose this language extension to focus here on the meta-level architecture. Section 3 shows how this same architecture is used to implement more complex mechanisms such as traits, Beta prefixing and multiple inheritance.

To implement such a behaviour in our architecture, we will add AutomaticAccessorPart, a subclass of ContributingPart doing the corresponding accessors generation. Figure 2 shows how this new extension follows our model and how a logical Person class imports it by using an instance of the contributing part. In this example, we implement the method calculateMethods(originalMethods) in AutomaticAccessorPart to create a getter and a setter method for each of the instance variables in the logical class.

A naïve implementation of such language extension (expressed in Pharo) is illustrated in Listing 1. We then return a collection containing the original and the new methods. A more robust implementation, out of the scope of this paper, would also take care of possible conflicts.

The listing shows the implementation of calculateMethods. This method is executed by the class builder to get the methods in the generated class. This method takes the collection of instance variables defined in the logical class, and per each of the variables it

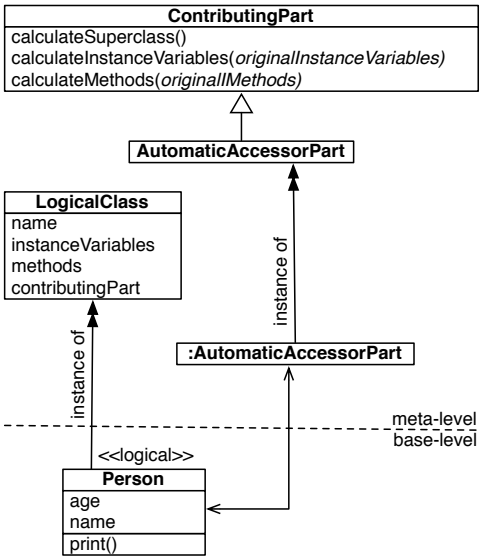

Figure 2: Implementing Automatic Accessors. The automatic accessor extension is implemented as a new contributing part. On the base-level, a Person logical class uses such extension.

```
AutomaticAccessorPart >> calculateMethods: originals
"Generates a new collection with the newly created methods
to be added to the runtime class"
 | newMethods |
 newMethods := logicalClass instanceVariables
  flatCollect: [ :instVar |
    { self createGetterFor: instVar.
    self createSetterFor: instVar}].
"Concatenates the original methods and the new ones".
 ^ originals , newMethods
```

Listing 1: Automatic Accessor Implementation. The method calculateMethods is overridden to provide new getter and setter methods for each of the logical class instance variables.

generates two methods (a setter and a getter). The generated methods are concatenated with the original ones, as none of the original methods are modified. To use this extension, a developer defines a logical Person class defining two instance variables age and name and using this as

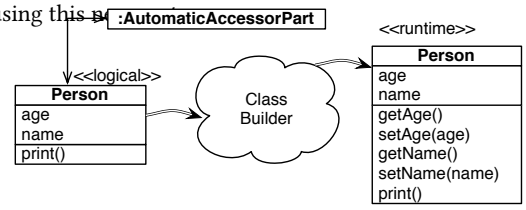

Figure 3: Resulting Runtime Class. The class builder generates a runtime class that contains the expected getters and setters in addition to the original methods and instance variables defined in the logical class.

Listing 2 illustrates how we can define such a class. Our language extension will automatically generate the methods getAge(), setAge(age), getName() and setName(name) in the runtime class. Figure 3 shows the resulting runtime class generated when the logical

```
Object
 subclass: #Person
 withParts: AutomaticAccessorPart
 instanceVariables: 'age name'

Person >> print
 StdOut print:(self name , ':' , self age , ' years')
```

Listing 2: Person Logical Class Definition. Code definition of the Person class. It contains only two instance variables and a print method.

```
AutomaticAccessorPart >> createGetterFor: instVar
  | sourceCode |
  sourceCode := instVar name , String newLine.
  sourceCode := sourceCode , '    ^' , instVar.name.
  ^ CreateMethod newWith: sourceCode

AutomaticAccessorPart >> createSetterFor: instVar
  | sourceCode |
  sourceCode := instVar name , ': aValue' , String newLine.
  sourceCode := sourceCode , '    ', instVar.name , ':= aValue'.
  ^ CreateMethod newWith: sourceCode
```

Listing 3: Creating the Operations for Accessors Method. AutomaticAccessorPart generates two CreateMethod operations that will compile the corresponding methods from the source code.

class uses the AutomaticAccessorPart, the generated runtime class contains the elements originally in the logical class Person and all the generated elements. In the following sections, we extend and explain in detail the MOP proposed by ContributingPart, and logical classes.

2.2 The Class Building Process

The ClassBuilder class reifies the building process transforming a logical class into a runtime class. It encapsulates all the low-level details of class creation, including the runtime representation of the runtime class and the bytecode used by the compiler.

To build a runtime class from a logical class, a class builder calculates the superclass, methods and instance variables that will compose such runtime class. For this, it calculates a set of *modification operations* that should be applied on the logical class to reach the runtime class definition. Such operations are calculated in collaboration with the class *contributing part*. The initial set of modifications of a class builder are the additions of the instance variables and methods locally defined in the logical class. In turn, the contributing part overrides, removes or adds new definitions to that set (c.f., Section 2.3). Finally, the resulting operations are applied to the new runtime class.

More concretely, the class builder invokes the contributing part calculateInstanceVariables and calculateMethods methods with the initial set of definitions as *addition operations*. In our motivating example, the AutomaticAccessorPart class generates a CreateMethod operation for each accessor, so the class builder compiles the accessor methods in the runtime class. The code of such change is illustrated in listing 3. The code shows how we could implement the methods createGetterFor(instVar) and createSetterFor(instVar) from the example in Listing 1.

2.3 Contributing Parts MOP

In our architecture, a reuse mechanism is modeled as a meta-object, a (sub-)instance of ContributingPart. When the class builder generates the underlying VM runtime model, a contributing part calculates the methods, instance variables and superclass of the generated class. Figure 4 shows the interface implemented by a ContributingPart. The contributing part includes the information needed to construct the runtime classes (*e.g.,* the superclass in single inheritance, or the traits in the Trait contributing part). Our architecture is modular: contributing parts do not need to be included by default in the programming language runtime. Indeed, they can also be packaged and loaded separately and as any other library or module. By default, our solution packages the single inheritance mechanism, explained in Section 3.

ContributingPart
calculateSuperclass() calculateInstanceVariables(*originalInstanceVariables)* calculateMethods(*originalIMethods)*

Figure 4: Contributing parts interface.

A runtime class is created by combining several first-class modification operations on methods and instance variables. The methods calculateInstanceVariables and calculateMethods receive as argument the set of original operations as provided by the logical class, and may return a new set by applying some modifications to it. We modelled such operations as first-class objects for two main purposes. First, first-class operations can be easily composed and extended with new operations. For example, a rename method is defined by wrapping the original method in a *rename method object*. Second, composing operations instead of directly modifying the original collection allows to more easily represent complex operations like flattening or aliasing, useful to model traits or beta-prefixing as we will show in Section 3. Figure 5 shows examples of such operations including installing, removing or renaming methods and instance variables.

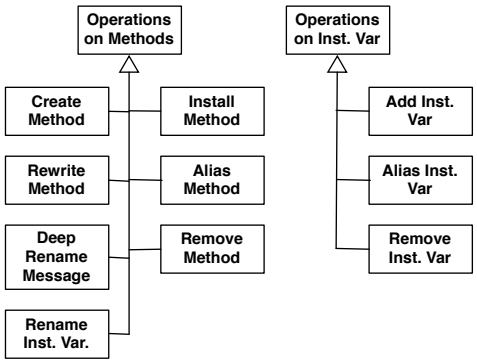

Figure 5: Operations on Methods and Instance Variables.

3 MODULAR REUSE MECHANISMS

In this section, we present how to implement different reuse mechanisms using this architecture. As a validation, we implemented four different reuse mechanisms as contributing parts: the default single inheritance mechanism, stateful traits, Beta prefixing and CLOS multiple inheritance. In the last example, we show how we can focus on performance of the generated code to demonstrate how such a mechanism can be optimized for a single inheritance VM.

3.1 Single Inheritance

In our architecture, a class using single inheritance contains only one contributing part, the single inheritance contributing part. The implementation of this contributing part in the runtime level is also the simplest one. It sets the real superclass to the configured superclass and does not add or modify any of the methods or instance variables. As the underlying implementation in the VM is already providing method lookup. The contributing part is not modifying the originals methods and instance variables. The calculateSuperclass() method returns the configured superclass in the contributing part.

This first reuse mechanism is straightforward because the semantics between the implemented reuse mechanism (single inheritance) and the semantics of the VM match.

3.2 Stateful Traits

This contributing part introduces support for stateful traits [5]. Compared to our previous example, this model does not directly match with the VM semantics. Our traits contributing part will flatten stateful traits in classes that use them. The contributing part is configured with a trait composition. A trait composition is a set of traits and operations defining how to resolve conflicts.

This part does not modify the superclass of the generated class, it keeps the one defined in the logical class. It adds all the instance variables and methods defined in the trait composition. It also handles the aliasing of methods and instance variables. The resulting runtime class contains all the methods and instance variables that are defined in the trait composition. The execution of the runtime code is transparent to the Virtual Machine, as the method lookup is already solved.

A trait composition [5] is an algebra implemented with objects used to solve the conflicts. It is able to answer the methods and instance variables that form the trait. The simplest possible one gets all the methods and instance variables of a given class. The trait composition is then used by the trait part to calculate the required methods and instance variables.

3.3 Beta Prefixing

In Beta, a *subpattern* is an extension of a previously defined *prefix* pattern [27]. It is possible to see the *subpatterns* as subclasses and the *prefix patterns* as superclasses. A developer defines new implementations in subclasses. However, in this reuse model the methods in the prefix cannot be modified, and every time a message is sent the prefix method is the one that is activated. The methods in the subpatterns are only activated if the prefix invokes them explicitly through the use of inner.

The Beta contributing part knows the prefix (superclass) to be used in the definition of the class, and this is the superclass used in the generated class. The part contributes with the methods and instance variables defined in the superclass. The instance variables

in the subclasses follow the same behaviour as the single inheritance model, so there is no need for special conversion.

For the methods only defined in the prefix or only defined in the subpattern, there is no need to execute anything special. They are defined in the corresponding class and they are normally resolved by the method lookup in the VM.

However, for the methods that redefines a prefix, the method lookup has to be modified. As the VM cannot be modified, the contributing part installs a stub in the generated class. This stub will lookup the method in the top-most class of the subclass hierarchy and activates this method on the receiving object. The method defined in the subpattern is also installed in the generated class but with an aliased name.

To implement the inner operand, the contributing part installs in the generated class the inner() and innerWithArgs(args) methods. These methods activate the following method in the subpattern chain, handling the de-aliasing of methods' name.

The implementation of the contributing parts should guarantee that the resulting runtime class does not have conflicting methods or instance variables. When two or more methods with the same selectors should be installed in the generated class, as is happening with the generated stub and the real method. One of the methods should be aliased with another selector. The architecture provides an operation, AliasMethod, to easily alias a method.

A more complex but more efficient alternative for avoiding the lookup stub is implemented in the next section on CLOS multiple inheritance contributing part.

3.4 CLOS multiple inheritance and Standard Method Combinations

CLOS implements multiple inheritance and execution of *after* and *before* methods, surrounding the normal methods (called *primary methods*) [25].

We implemented a contributing part that provides the same behaviour in the generated classes. This contributing part also implements the *call-next-method* mechanism, following the rules of inheritance and *before* and *after* methods. CLOS also implements the *around* methods, that are called before the *before* methods. For implementing *around* methods, the same implementation technique is applicable. For the clarity of the explanation we will only present *before*, *primary* and *after* methods.

The inheritance in CLOS is linearised using an algorithm to avoid the diamond problem [19]. Our contributing part implements this same strategy.

The contributing part calculates the expected execution chain methods for each of the messages in a class. The chain is calculated following the rules expressed in the definition Standard Method combination of CLOS.

Synthetically, these rules are:

- Primary methods form the main body of the effective method. Only the most specific primary method is called. However, the method is able to call the next most specific primary method by doing (call-next-method).
- *Before* methods are all called before the primary method, with the most specific *before* method called first.
- *After* methods are all called after the primary method, with the most specific *after* method called last.

After calculating the method chain to execute for a given message the contributing part performs the following steps:

(1) It takes the first method in the chain, and it aliases the method with the expected message selector. So the VM will activate this method when an object receives the expected message.
(2) It aliases all the other methods in the chain, so they do not conflict with each other, as they have the same message selector.
(3) It includes a nextWithArgs(args) method used to call the next method in the chain. The next method in the chain is activated from the information already calculated.
(4) It rewrites all the calls to nextWithArgs(args) with the corresponding call to the calculated next method, using the aliases selectors.

Figure 6 shows the definition of a logical class Dog. This class extends Mammal and Quadruped, giving Mammal precedence over the other superclass. This definition is performed using the CLOS contributing part. For this example, we will suppose that all the defined methods have a *call-next-method* operation. If the message walk() is sent to an instance of Dog, the expected activation of methods is the following: before walk() in Dog, before walk() in Mammal, before walk() in Quadruped, walk() in Dog, walk() in Mammal, walk() in Quadruped, after walk() in Quadruped, after walk() in Dog.

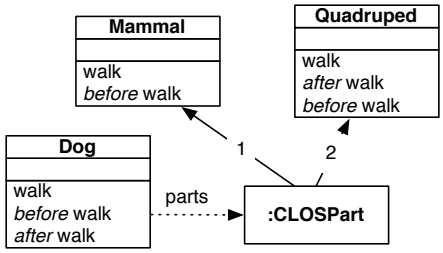

Figure 6: The Dog class with its contributing part, implementing CLOS-like multiple inheritance

Figure 7 depicts the resulting generated classes thanks to our CLOS contributing part. In the generated Dog class, the first method in the chain (before walk() in Dog class) is aliases as walk(), as it will be the first method activated by the VM. The other methods are installed with aliases selectors, to avoid name clashes.

The instance variables are collected from all the superclasses and combined with the locally defined instance variables. For the instance variables, our current implementation does not handle the conflicts, although the conflict is solvable using instance variables aliasing. The implementation creates the runtime classes with Object as the superclass. As all the inheritance mechanism is handled as explained before.

The implementation of Mixins is simpler than the CLOS multiple inheritance mechanism. However, it uses the same techniques used in this implementation of CLOS multiple inheritance. It is also possible to use this approach to implement the C3 linearisation algorithm used in Dylan [3].

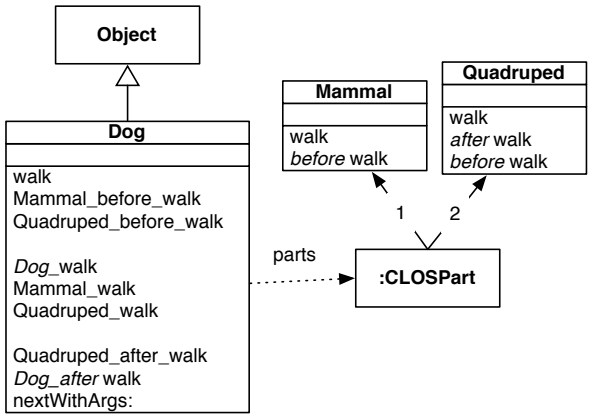

Figure 7: The runtime representation of the Dog class

4 COMPOSING REUSE MECHANISMS

Our proposed architecture allows the composition of the reuse mechanisms. In this way the developer is able to use different reuse mechanisms at once. This feature improves the modularity of the solution and provides easier paths for its extension. The composition of contributing parts is performed through the implementation of a Chain of Responsibility pattern [21].

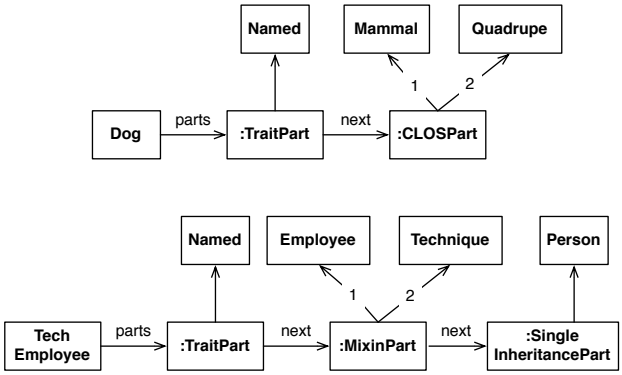

Figure 8: Combining different mechanisms.

The logical classes have a chain of reuse mechanisms. This chain is used to calculate the elements in the generated class. Each contributing part in the chain can contribute and/or pass the control to a subsequent part to generate the runtime classes, *e.g.,* adding, filtering, or changing new elements. The generated runtime classes contain the methods and instance variables produced by this chain. Each class defined in the system has its own unique chain of contributing parts, allowing them to have different combination of mechanisms. Figure 8 shows how different contributing parts are combined in different classes.

To help implementing the combining reuse mechanisms, we provide an abstract implementation of a contributing part. This AbstractContributingPart implements the handling of the chain of responsibility and provide default implementations for the required methods. Figure 9 shows how the different contributing parts extend the AbstractContributingPart with their custom behaviour.

Even though, the proposed architecture allows the arbitrary composition of reuse mechanisms. Some of the possible combinations could be invalid. We have successfully implemented combinations of the four reuse mechanisms (Section 6.1).

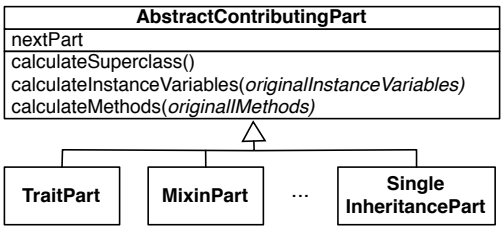

Figure 9: Introducing AbstractContributingPart that implements the chaining mechanism

5 MODULAR REFLECTIVE MOP

Each class in the *logical level* is defined by (a) a set of elements defined in the class (methods and instance variables) and (b) a configured contributing part. The instance variables and methods defined in a runtime class depend on (1) the elements defined in its corresponding logical class and (2) the reuse mechanism used in this class. To allow this distinction at runtime, by the different programs using reflection, we propose also a modular reflective MOP. Logical classes expose a reflection MOP that allow us to modify the class in terms of the code-reuse mechanisms used in the language. On the other hand, runtime classes expose a MOP that allow us to modify their low level representation. In practice, logical and runtime classes can be seen as high-level and low-level mirrors [12]. The main difference between our reflective MOP and the one of mirrors is that we designed ours with modularity in mind.

In our model, we can then define for each logical class two sets of elements, the *local elements set* and the *all elements sets*. The *local elements set* includes all the elements (methods and instance variables) that are defined in the class itself. The *all elements sets* includes the result of applying the contributing part to the class. This MOP provides a polymorphic way of interacting with all the logical classes, hiding the details of how the contributing parts calculate the elements in the runtime class. Continuing with the example in Figure 3, the local elements set of Person contains the print() method and the age and name instance variables, as they are defined locally in the logical class. And the all elements set contains the locally defined elements elements (print(), age and name) and the elements provided by the used contributing part (AutomaticAccessorPart), those elements are getAge(), setAge(age), getName() and setName(name).

Our defined MOP does allow us to query and modify the logical model. Each modification in the logical class triggers a regeneration of the runtime model according to the changes done. The logical MOP includes the following set of messages:

- **Local Elements Query.** Access information about the elements defined in the logical class.
- **All Elements Query.** Access information about all the elements in a logical class. It includes all the locally defined elements and the contributed ones.
- **Modify Local Elements.** Addition, removal and modifications of the locally defined elements.
- **Accessing Contributing Parts.** Access and modification a contributing part.
- **Regenerating class.** Regeneration of the runtime class, impacting any changes performed in the logical class.

The contributing parts are also subject to modifications during the execution of the system. They also present an interface to get all the elements contributed to a logical class, and they also provide messages to modify its configuration. Any modification in the contributed part requires the execution of the *regenerating class* in the modified logical class.

6 VALIDATION

We validated our architecture by implementing it in the Pharo [7] programming language. This implementation served us to evaluate how several contributing parts are implemented and combined.

Our implementation is available as a Github project and it runs in Pharo 6. This implementation includes all the mechanisms described in this paper as well as a set of tests and benchmarks combining these mechanisms. We selected Pharo because: (1) it runs in a highly optimized VM that expects a single inheritance model, (2) it provides an easy modification of the class creation process, (3) it represents basic concepts of the language and the environment as first-citizen objects and (4) most of the infrastructure is implemented in itself. These characteristics ease the implementation of our modular solution as an independent and regular package.

6.1 Evaluating Part Composition

Even though the proposed architecture allows the arbitrary composition of reuse mechanisms, some of the possible permutations[2] could indeed be invalid. It is outside the scope of this paper the analysis of the validity of possible combinations of reuse mechanisms. However, we have successfully implemented the following combinations:

- **Traits / Single Inheritance**. Extending classes using traits or using as traits classes that are part of a single inheritance hierarchy.
- **Traits / CLOS Multi inheritance**. Using classes with traits as superclasses or using multi inheritance classes as traits.
- **CLOS Multi inheritance / Single Inheritance**. Having a single inheritance class extending a multi inherited class and the other way around.
- **Traits / Beta prefixing**. Using classes with traits as prefixes, adding prefixes and traits to the same class and using beta's classes as traits.

6.2 Benchmarks

The architecture and the implemented contributing parts have been validated to assure that they do not introduce differences in the execution time. We have benchmarked different scenarios, but only one of them introduces an impact in the execution time once the runtime classes are generated. The scenario is when the *super-like message sends* are overridden. It occurs in the CLOS and Beta implementation.

We have detected that the naïve implementation in Beta's *inner* is 100 times slower than a normal super message send. However, a more carefully implemented solution of it does not present this problem. The implementation of CLOS *call-next-method* is comparable to a normal super call.

[2] since contributing parts are organized as a chain of responsibility, order is important

7 DISCUSSION

Our solution allows us to introduce new reuse mechanisms, to extend the ones defined in the language, to develop new mechanisms and load them as a library. With this solution, it also possible to combine different mechanisms and use them in different parts of the application.

Our solution relies on a contributing parts model to easily implement modular reuse mechanisms thanks to different elements to represent the changes on the instance variables and methods such as: renaming, removing and adding methods. Reifying those elements make the implementation high-level and simpler. Indeed, having the class building operations internally to the architecture and letting only expressing the changes through objects make it easier implementing reuse mechanisms because details are hidden from their contributing parts. Meta programmers are able to extend the existing modifications to elements, providing more flexibility to implement reuse mechanisms.

Our solution does not permit the generation of more classes. The mechanisms only generates one runtime class per class in the logical level. We decided so, to keep the runtime model similar to the logical model. When inspecting the objects, the classes of these objects are the same as the ones in the logical model.

To achieve performance comparable to the execution of single inheritance code, our solution duplicates methods. This is performed also when the method lookup is overridden for a given mechanism. This optimization produces bigger runtime programs (memory overhead) but the performance is not affected.

When a class or a method is modified in the logical model, the generated classes should be updated, making the compilation longer. Because the architecture keeps the logical and runtime models in sync. However, we decided to do it in this way as the dynamic modification of the program is less frequent compared with the execution of the methods.

8 RELATED WORKS

Kotlin[3], Scala, Groovy and Jython provide high performance implementations of the reuse mechanisms presented in this paper on top of the Java Virtual Machine through the direct generation of bytecode. However, their implementation does not allow the modification of these reuse mechanisms or the loading of new mechanisms as a library module. The Traits implementation existing in Pharo does not allow the modification of reuse mechanism without the modification of the core classes [32].

Cazzola et al. [13] propose a modular implementation for the development of interpreters. By the modification of the interpreter is possible to modify the reuse mechanisms existing in a given language. However, this technique requires the development of low level interpreters. Also the resulting code executes in the developed interpreters, without taking advantage of the underlying JVM. As in our solution, the custom interpreter implementation is very sensitive to the performance of the execution. Although having to implement a lower level mechanism requires a deeper understanding of the underlying execution environment.

Bouraqadi et al. [8, 9] show how classes can be extended in a reflective language using metaclasses. Metaclasses can then be

[3] https://kotlinlang.org/

used as an extension mechanism to add new properties to class in a per-class basis. While the Metaclass proposal is indeed good, it mixes both the low level representations of classes with the high level representations. In our meta-level architecture there is a clear separation of these concepts, what we believe is necessary for maintenance purposes. Bettini et al. [6] show how to statically extend Java with traits by static generation of code. This approach has the advantage of providing full IDE integration. Composition filters [1] and MixDecorator [30] also propose variations of GOF design patterns to change the behaviour of a program. We chose instead to use a chain of responsibility pattern, hiding from the user the complexity of combinations and identify of objects. Finally, our model proposes the creation and combination of reuse mechanisms but it is out of the scope of this paper to work on the validation of the possible combinations, as done in GenVoca [4].

9 CONCLUSION

In this paper, we propose a novel architecture for the modularization of reuse mechanisms. This architecture allows the developer to work with different reuse mechanisms, taking advantage of the different benefits of each of them. Also, it allows meta programmers to implement new reuse mechanisms and combine them in novel ways without the need to modifying the underlying VM. The architecture is intended to generate classes to be run in a high-performance single inheritance VM as the Java Virtual Machine or the Cog Smalltalk Virtual Machine.

Even though our solution allows a high level of flexibility. It allows to be extend in different ways (from the contributing parts to the modifications of methods and instance variables). It still requires optimized implementations of the reuse mechanisms if the performance is a must-have.

As future work, we are interested in the detection of patterns in the contributing parts and common problems of performance. Also we are interested in how this runtime and logical level MOP interacts with the tools in the IDE.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

REFERENCES

[1] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. 1993. Abstracting Inter-Object Communications Using Composition Filters. (1993). University of Twente.

[2] Matthew Arnold, Stephen J Fink, David Grove, Michael Hind, and Peter F Sweeney. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE* 93, 2 (2005), 449–466.

[3] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. 1996. A Monotonic Superclass Linearization for Dylan. In *Proceedings OOPSLA '96, ACM SIGPLAN Notices*. 69–82.

[4] Don Batory and Bart J. Geraci. 1997. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)* (Feb. 1997), 62–87. http://www.cs.utexas.edu/users/schwartz/

[5] Alexandre Bergel, Stéphane Ducasse, and Lukas Renggli. 2008. Seaside – Advanced Composition and Control Flow for Dynamic Web Applications. *ERCIM News* 72 (Jan. 2008). http://ercim-news.ercim.org/content/view/325/536/

[6] Lorenzo Bettini. 2016. *Implementing domain-specific languages with Xtext and Xtend.* Packt Publishing Ltd.

[7] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. 2009. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland. 333 pages.

[8] Noury Bouraqadi. 2004. Safe Metaclass Composition Using Mixin-Based Inheritance. *Journal of Computer Languages, Systems and Structures* 30 (April 2004).

[9] Noury Bouraqadi, Thomas Ledoux, and Fred Rivard. 1998. Safe Metaclass Programming. In *Proceedings OOPSLA '98.*

[10] Don Box and Ted Pattison. 2002. *Essential. NET: The common language runtime.* Addison-Wesley Longman Publishing Co., Inc.

[11] Gilad Bracha and William Cook. 1990. Mixin-based Inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, Vol. 25. 303–311.

[12] Gilad Bracha and David Ungar. 2004. 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.* ACM Press, New York, NY, USA, 331–344. http://bracha.org/mirrors.pdf

[13] Walter Cazzola and Albert Shaqiri. 2017. Open Programming Language Interpreters. *The Art, Science, and Engineering of Programming* Vol.1 (2017). Issue 2.

[14] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. 1997. Compiling Java just in time. *IEEE Micro* 17, 3 (1997), 36–43.

[15] O.-J. Dahl, B. Myrhaug, and K. Nygaard. 1970. *(Simula67) Common Base Language.* Technical Report N. S-22. Norsk Regnesentral (Norwegian Computing Center), Oslo, N.

[16] Linda G. DeMichiel and Richard P. Gabriel. 1987. The Common Lisp Object System: An Overview. In *Proceedings ECOOP '87 (LNCS)*, J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman (Eds.), Vol. 276. Springer-Verlag, Paris, France.

[17] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. 2006. Traits: A Mechanism for fine-grained Reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 28, 2 (March 2006), 331–388. https://doi.org/10.1145/1119479.1119483

[18] R. Ducournau and Michel Habib. 1987. On Some Algorithms for Multiple Inheritance in Object-Oriented Programming. In *Proceedings ECOOP '87 (LNCS)*, J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman (Eds.), Vol. 276. Springer-Verlag, Paris, France, 243–252.

[19] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. 1992. Monotonic Conflict Resolution Mechanisms for Inheritance. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, Vol. 27. 16–24.

[20] David Ehringer. 2010. The dalvik virtual machine architecture. *Techn. report (March 2010)* 4 (2010), 8.

[21] Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings ECOOP '93 (LNCS)*, Oscar Nierstrasz (Ed.), Vol. 707. Springer-Verlag, Kaiserslautern, Germany, 406–431. ftp://st.cs.uiuc.edu/pub/papers/patterns/ecoop93-patterns.ps

[22] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings ECOOP '91 (LNCS)*, P. America (Ed.), Vol. 512. Springer-Verlag, Geneva, Switzerland, 21–38.

[23] Cheng-Hsueh A Hsieh, John C Gyllenhaal, and Wen-mei W Hwu. 1996. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture.* IEEE Computer Society, 90–99.

[24] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto Muoz, Victor Ng, Alex Ng, and Donna L Baker. 2010. *The definitive guide to Jython: Python for the Java platform.* Apress.

[25] Sonia E. Keene. 1989. *Object-Oriented Programming in Common-Lisp.* Addison Wesley.

[26] Dierk König. 2007. *Groovy in action.* Manning.

[27] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. 1987. The BETA Programming Language. In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner (Eds.). MIT Press, Cambridge, Mass., 7–48.

[28] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification.* Pearson Education.

[29] Eliot Miranda. 2011. The Cog Smalltalk Virtual Machine. In *Proceedings of VMIL 2011.*

[30] Virginia Niculescu. 2015. MixDecorator: An Enhanced Version of Decorator Pattern. In *Proceedings of the 20th European Conference on Pattern Languages of Programs (EuroPLoP '15).* ACM, New York, NY, USA, Article 36, 12 pages. https://doi.org/10.1145/2855321.2855358

[31] Martin Odersky. 2007. *Scala Language Secification v. 2.4.* Technical Report. École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland.

[32] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. 2003. Traits: Composable Units of Behavior. In *Proceedings of European Conference on Object-Oriented Programming (LNCS)*, Vol. 2743. Springer Verlag, 248–274.

[33] Michel Schinz. 2005. Compiling Scala for the Java virtual machine. *EPFL,* (2005).