An Experiment with lexically-bound extension methods for a dynamic language

Stéphane Ducasse, Luc Fabresse, Guillermo Polito, Camille Teruel

Tech Report: An Experiment with lexically-bound extension methods for a dynamic language

Stéphane Ducasse [*1], Luc Fabresse [†2], Guillermo Polito [‡3], and Camille Teruel [§1]

[1]Inria Lille Nord Europe - Cristal Université de Lille
[2]Mines Douai, Mines-Telecom Institute
[3]Inria Lille Nord Europe - Cristal Université de Lille

February 21, 2017

Abstract

An extension method is a method declared in a package other than the package of its host class. Thanks to extension methods, developers can adapt classes they do not own to their needs: adding methods to core classes is a typical use case. This is particularly useful for adapting software and therefore increasing reusability.
In most dynamically-typed languages, extension methods are globally visible. Because any developer can define extension methods for any class, naming conflicts occur: if two developers define an extension method with the same signature in the same class, only one extension method is visible and overwrites the other. Similarly, if two developers each define an extension method with the same name in a class hierarchy, one overrides the other. Existing solutions typically rely on a dedicated and slow method lookup algorithm to resolve conflicts at runtime.
In this article, we present a model of scoped extension methods that minimizes accidental overrides and we present an implementation in Pharo that incurs little performance overhead. This implementation is based on lexical scope and hierarchy-first strategy for extension scoping.

[*]stephane.ducasse@inria.fr
[†]luc.fabresse@mines-douai.fr
[‡]guillermo.polito@inria.fr
[§]camille.teruel@gmail.com

1 Conch

We present a model named Conch that combines lexical activation and hierarchy-first strategies to limit accidental overrides as demonstrated by previous analysis [6]. We believe that this solution is close to the *selector namespace* mechanism of the *SmallScript* language, but we could not find either an article or the code to precisely verify it. We present in a following section a prototype of Conch implemented in the Pharo programming environment.

1.1 Conch Model

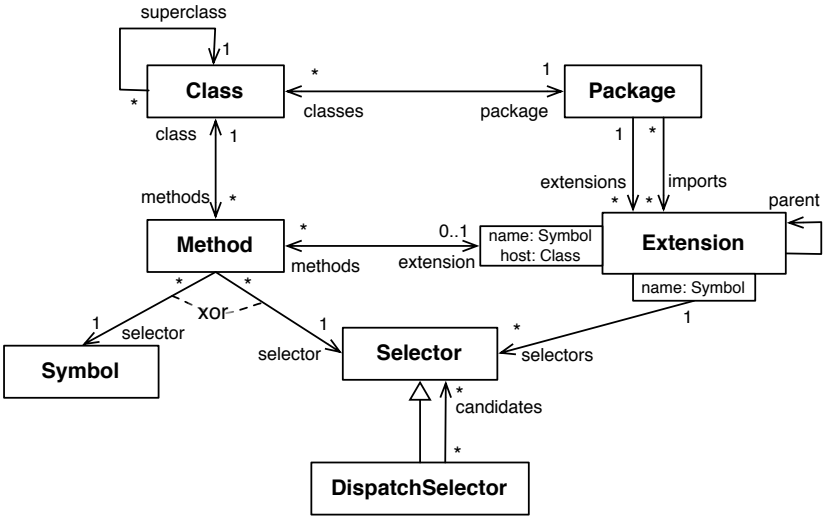

Figure 1: The Conch model.

Figure 1 presents an overview of the Conch model.

- Packages contain classes and extensions.
- An extension can be imported at the package, class or method level.
- An extension indexes the extension methods it contains by name and host class.
- An extension can have a parent extension.
- The child extension can declare methods that are polymorphic with the one declared in its parent: *i.e.,* the messages sent by client code that uses the parent extension can dispatch to methods declared in the child extension.
- A child extension cannot override an extension method declared in its parent extension.

1.2 Conch by Example

Extension methods are useful when one wants to enhance an external package with new methods. Let's consider a parser combinator package named PetitParser. We can consider creating an extension to this package providing a DSL of predicates to match expressions upon various criteria. This compatibility package declares a extension of the PetitParser package (which declares the asParser extension methods). A child extension may declare its own version of asParser in predicate classes to convert predicate objects into parsers. Client code that imports the PetitParser extension can then use the extension of the compatibility package without changes.

2 Conch Implementation in Pharo

This implementation of Conch[1] has been built with certain constraints in mind. First, it is dynamic typing compliant. Indeed, the absence of static type information prevents the compiler to know the receiver class of a message send. Consequently, the message may dispatch to an extension method or to an unrelated class. Second, since extension methods are widely used in Pharo, we want as little performance loss as possible. Third, the migration to the this new method extension model must be feasible without rewriting every existing Pharo package from scratch. Finally, this implementation relies on the default method lookup of Pharo instead of changing it in the virtual machine.

Before presenting some of the internals of its prototype, we first briefly present the Pharo method lookup inherited from Smalltalk.

2.1 Smalltalk method lookup

As a single-dispatch language, this lookup doesn't take the class of the arguments into account. Also the number of arguments is encoded into the method name, that is called a selector. Hence a method signature consists solely of a selector. Selectors are globally unique strings (instances of the class Symbol). The methods of a class are stored in a dictionary that maps method selectors to method objects. When a message with a given selector is sent to an object, the lookup algorithm searches for the associated method as follows. First, the current class is set to the class of the receiver object. If the method dictionary of the current class contains a method for the given selector, the lookup stops and returns that method. Otherwise, the lookup pursues in the current class' superclass. If no method is found in the hierarchy, the message #doesNotUnderstand: is sent to the receiver object with a reification of the first message as argument. Overriding #doesNotUnderstand: permits instances to answer to any message but proxies frameworks such as [5] are much more complete.

[1] Available at `http://smalltalkhub.com/\#!/~CamilleTeruel/ScopedExtensions`

2.2 Selector mangling implementation

The key point of this implementation is to distinguish a method name from its selector [8]. The name is what the developer types in source code while the selector is the object used to look into class method dictionaries at runtime. An extension is akin to a namespace: it maps a method name to a unique selector object. Instead of Symbol the selector of scoped methods are instances of a class Selector. Symbols are still used as selectors for regular methods.

Before a method m is compiled, its abstract syntax tree (AST) is transformed. The transformation changes the selector of the AST method node from a symbol to the associated selector object of m's extension e. If e has a parent extension, the parent's selector associated with m's name is used. If no selector object is associated with m's name in e or its parent (*i.e.,* m is the first method of these two extensions with this name), a new selector object is generated and installed in e.

Each AST message node selector is then transformed as follows. Each extension that is visible from the method is queried for the selector associated with the message name in order of priority. Because several extensions can declare extension methods with the queried name, this yields an ordered list of selectors objects. If this list contains only one selector object, we replace the name of the message node (a symbol) with that selector object. If this list contains several selectors objects, we replace the name of the message node with a special selector object, instance of the class DispatchSelector. Dispatch selectors are used to resolve dynamically the ambiguities that exist at compile-time. The dispatch selector takes the list of selector objects to be looked-up at runtime. To implement the hierarchy-first method selection strategy, each selector of this list must be looked-up one by one in the receiver class hierarchy until a method is found.

2.3 Self/super optimization

In case the message node is a self send, we know that the method lookup will start in the class of the receiver. Consequently, we only consider the selector objects of extensions that define an extension method for either the class of the receiver, one of its superclass, or one of its subclasses. In case the message send is a super send, we know that the method lookup will start in the superclass of m's class. Consequently, we only consider the selector objects of extensions that define an extension method for either the superclass of the class of m, or one of its superclasses.

2.4 Backward compatible error handling hook.

Normally, when the method-lookup fails the virtual machine sends the message #doesNotUnderstand: to the receiver with a reification of the message passed as an argument. We use a selector object (*i.e.,* not a symbol) #retry: instead to

preserve the behavior of classes that override #doesNotUnderstand:. A method with this same selector object #retry: is defined in ProtoObject:

```
ProtoObject>>retry: aMessage
    ↑ aMessage selector retryFor: self withMessage: aMessage
```

The #retryFor:withMessage: method of the class Symbol sends the normal #doesNotUnderstand: message:

```
Symbol>>retryFor: anObject withMessage: aMessage
    ↑ anObject doesNotUnderstand: aMessage
```

This permits classes that redefine #doesNotUnderstand: to behave as intended. The method #retryFor:withArguments: of the class Selector is defined as follows:

```
Selector>>retryFor: anObject withMessage: aMessage
    (anObject class lookupSelector: self name) ifNotNil: [ :method |
        self installAliasesFor: method.
        ↑ anObject perform: selector withArguments: aMessage arguments ].
    ↑ anObject doesNotUnderstand: aMessage
```

This method first checks if the receiver class implements or inherits a regular method that has the same name. This is how we implement the implicitly imported global extension. If a method is found, method aliases are installed: for the found method and for each overriding method, a new binding that associates the selector object with the method is added into the method dictionary of that method class. Consequently, the next lookups of this selector in that class hierarchy will succeed directly and be as fast as normal message send.

For dispatch selectors, the situation is similar. At runtime, when a message with a dispatch selector is sent for the first time, the lookup will fail because no method has this dispatch selector as selector. The method #retryFor:withMessage: of the class DispatchSelector is defined as follow:

```
DispatchSelector>>retryFor: anObject withMessage: aMessage
    self selectors do: [ :each |
        (anObject class lookupSelector: each) ifNotNil: [ :method |
            self installAliasesFor: method.
            ↑ anObject perform: selector withArguments: aMessage arguments ] ].
    ↑ super retryFor: anObject withMessage: aMessage
```

This method searches the first selector understood by the receiver's class and sends it to the receiver with the message arguments using the reflective send method #perform:withArguments: after installing the corresponding method aliases. If none of the selector is understood by the receiver's class we fallback with Selector's behavior. This is how the hierarchy-first method selection strategy is implemented.

2.5 Summary

Conch is an implementation of lexically-scoped extensions methods for Pharo that requires no modification of the virtual machine. It solves the problem of ambiguous call-sites thanks to dispatch selectors. Thanks to method aliases, it has a low performance overhead at runtime.

3 Related Work

Solutions in Statically-Typed Languages. In this paper, we have limited our research to solutions in dynamically-typed languages. There exists other solutions in the context of statically-typed languages. In C#, extension methods are essentially syntactic sugar over static methods whose first argument type is the extended type. Scala supports a mechanism similar to extension methods with *implicit classes*. These solutions are not portable to dynamically-typed languages because they rely on static type information.

MultiJava [1] is a Java [3] extension that support open classes and multiple dispatch. The open-class solution proposed by MultiJava is close to a proposition presented in Section 1 for dynamic languages. MultiJava protects from accidental overrides: a method m overrides an extension method m' only if m' is imported in the file that declares m.

Expanders [7] are another language construct that support scoped extension methods in the context of *eJava*, a Java extension. An expander is a class extension that can be brought into the lexical scope of a compilation unit. It allows classes to be updated with new methods, fields and interfaces. An expander can override the extension methods defined in another expander. This solution enables intended overrides while preventing accidental ones. However, an extension method cannot override a regular method. By importing expanders, client code adapts some classes to its particular need. Expanders rely on static type information and can be typed-checked modularly. Conch is an equivalent of expanders for dynamically-typed languages.

Module Refinement. PRM is a statically-typed language supporting class refinement [2]. In PRM, a module is a reuse unit supporting separate compilation that contains a class hierarchy. On the contrary, classes are no longer reuse units and there is no class nesting. A module depends on a set of other modules (called supermodules) that can refine classes imported from them. A submodule is not included in its supermodule: there is no module nesting either. Class refinement can be one of the four atomic mechanisms: (1) adding a property, i.e., the definition of a newly introduced method or attribute; (2) redefining (aka overriding) a property; (3) adding a superclass; (4) generalizing a property, i.e., defining a property in superclasses of the class which introduced it in the supermodules. The redefinition of a property is visible to the module that imports it and it relies on static typing to disambiguate its use.

Accidental overrides. Simple changes can have unexpected effects due to implicit contracts between a class and its subclasses. This well-known problem, coined as the *fragile base class problem* [4], is due to the fact that current languages do not support well extension contracts: Just changing the calling structure of a method without changing its external behavior may have unexpected effects in presence of subclasses. C# is the one of the rare languages that offers a way to control unintended name capture (called accidental overrides in this paper). C# allows the programmer to qualify a method with the keyword new (rather than override) to declare that while the newly defined method has the same name as the one in its superclasses, it is used for a different concept than in the superclasses. As such, all calls in the superclass hierarchy that would invoke a method with the same name will not consider the new method.

4 Conclusion

Globally-visible extension methods can lead to conflicts: accidental overrides and overwrites. These conflicts pose class encapsulation problems that can lead to subtle bugs or be exploited by malicious parties. In this article, we propose Conch, a solution for scoped extension methods that follows these conclusions, and described its implementation for Pharo that incurs little performance overhead by leveraging a traditional class-based method lookup algorithm. Its implementation is based on first-class and unforgeable selectors.

References

[1] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000*, pages 130–145, 2000.

[2] R. Ducournau, F. Morandat, and J. Privat. Modules and class refinement: a meta-modeling approach to object-oriented programming. Technical Report 07-021, LIRMM, Université Montpellier II, 2007.

[3] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[4] L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *ECOOP'98*, number 1445 in LNCS, pages 355–383. Springer-Verlag, 1998.

[5] M. M. Peck, N. Bouraqadi, L. Fabresse, M. Denker, and C. Teruel. Ghost: A uniform and general-purpose proxy implementation. *Journal of Object Technology*, 98:339–359, 2015.

[6] C. Teruel. *Adaptability and Encapsulation in Dynamically-Typed Languages: Taming Reflection and Extension Methods*. PhD thesis, Université Lille 1 - France, Jan. 2016.

[7] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaptation with expanders. In *OOPSLA '06*, pages 37–56, 2006.

[8] A. Wirfs-Brock. Subsystems — proposal. OOPSLA 1996 Extending Smalltalk Workshop, Oct. 1996.