# Automatic Reuse through Implied Methods

The Design and Implementation of an Abstraction Mechanism for Implied Interfaces

Patrick Rein
Hasso Plattner Institute, University of Potsdam
Potsdam, Germany
patrick.rein@hpi.uni-potsdam.de

## CCS CONCEPTS

•**Software and its engineering** →**Procedures, functions and subroutines;** *Modules / packages;*

## KEYWORDS

implied methods, generic operations, modules, reuse, planning, dispatch

**ACM Reference format:**
Patrick Rein. 2017. Automatic Reuse through Implied Methods. In *Proceedings of Programming '17, Brussels, Belgium, April 03-06, 2017,* 3 pages.
DOI: http://dx.doi.org/10.1145/3079368.3079371

## 1 INTRODUCTION

Object-oriented systems enable code reuse through inheritance. While multiple inheritance is a complex mechanism, single inheritance mechanisms limit code reuse as they only allow for reusing the implementation of a single class [2, 12]. As a result, a group of mechanisms have been proposed, commonly referred to as mixins [2, 4, 7, 12]. Mixins capture the definition of state, behavior, or both. To reuse the code of a mixin, developers have to explicitly apply it to a class.

However, developers might not always be able to explicitly apply a mixin to a class. For example, developers maybe want to extend the interface of objects created by a library. In such a case, developers sometimes do not have access to the library code or do not want to change the class definitions in the library code as the modifications complicate updates to the library. Another situation, in which an explicit extension of a class is not possible, is when developers deal with plain data objects with no explicit class. Such objects might result from an application allowing users to import data from unanticipated sources.

For some methods, an explicit application of a mixin is not necessary. Such methods are direct deductions of more basic methods. For example, between:and:, which checks whether an Object is within an interval, is directly expressed in terms of <=:

```
between: min and: max
  ^ min <= self and: [self <= max]
```

This method is generally applicable to any object that implements <=, given that the implementation matches the semantics of an ordering operator expected by between:and:. Other examples of such methods include the collection protocol ( select :, collect :, and similar) or geographical functions based on latitude or longitude ( greatCircleDistance or ellipsoidalDistance ).

Based on this observation we propose an abstraction mechanism which captures these *implied methods*. Based on the explicit definition of implied methods and their required interface, we also propose an automatic mechanism to dynamically extend the interface of suitable objects with these methods. As a result, implied methods separate code specific to domain objects (the implementation of <= which is specific for each class) from code which is concerned with the dependencies between general operations (the implementation of between:and:). Thereby, implied methods can make it easier to make interfaces of heterogeneous libraries compatible to each other.

## 2 IMPLIED METHODS

*Implied methods* consist of two parts:

(1) The method implementation to be provided, for example between:and:.
(2) A set of conditions the object has to comply with so that the implied method can be applied. In our example, this set only includes the condition that the object implements <=.

To make sure that the object provides the behavior needed for the implied method, the conditions can check for more than mere availability of method names. For example, to provide methods applicable to email addresses, the conditions might check whether an object is a string and whether its contents match a regular expression for an email address (see listing 1). This extended set of powerful conditions is necessary to distinguish between ambiguous method names, for example the message <= might also refer to a constructor for an association from right to left.

Generally, an implied method becomes applicable when all conditions are met by an object. The method can then become available in the interface of the object. As the conditions of an implied method might require other implied methods, the activation uses a planning algorithm to search for a suitable combination of existing implied methods.

Implied methods can be applied either in the case of a failed dispatch or pro-actively, for example at the interface boundary between a system and a library. Currently, the augmentation of objects with implied methods is dynamically scoped to not pollute the base system.

**Listing 1: Squeak/Smalltalk example of implied methods (#localPart, #domain) extending a String object which represents an email address. The condition of these implied methods is the runtime content of String objects.**

```
EMailString class>>#emailRegex
 ^ '([a-zA-Z0-9.]+)@([a-zA-Z0-9\-.]+)'

EMailString class>>#condition: object
 ^ object isString and: [object
  matchesRegex: self emailRegex]

EMailString>>#localPart
 (self allMatches: self class emailRegex)
  subExpression: 1

EMailString>>#domain
 (self allMatches: self class emailRegex)
  subExpression: 2
```

## 3 IMPLEMENTATION AND OBSERVATIONS

We have implemented implied methods in Squeak/Smalltalk [8], to evaluate the impact on the interactive nature of the environment, and in Racket/Scheme [5], to evaluate consequences resulting from a functional language. In both implementations implied methods are grouped by their set of conditions. In Squeak/Smalltalk all implied methods with the same set of conditions are described in one class (see listings 1, 2). In Scheme all implied methods are multi-methods sharing the same "conditions class" for their first argument. Both implementations currently extend the interface dynamically. To preserve any explicitly defined behavior, such as ordinary methods defined on the class of the object, implied methods are dispatched last.

By using implied methods with the Squeak/Smalltalk standard library, we were, for example, able to augment Point and String objects with the between:and: message. Further, through defining some intermediary implied methods we were also able to use one interface on objects representing address information from three sources.

**Listing 2: Squeak/Smalltalk example of an implied method #fullName extending objects which understand #firstName and #lastName.**

```
FullName class>>#condition: object
 ^ object understands: #firstName
  and: [object understands: #lastName]

FullName>>#fullName
 ^ self firstName asString , ' '
  , self lastName asString
```

## 4 RELATED WORK

Implied methods address the issue of flexible and fine-grained reuse in object-oriented systems. As they also provide a set of methods from outside of the single inheritance hierarchy they are related to the idea of Traits [4]. Further, implied methods are a mechanism to extend existing classes without access to their code. In this regard it is similar to many extension mechanisms like C# extension methods [11], GO interfaces [16], and open classes [1, 3]. Scala implicit classes even enable automatic extensions [14]. However, all these approaches require developers to explicitly extend a particular class with the new functionality. Further, most of them require the extension to be bound to a class and not to objects and their individual properties.

Pointcuts in aspect-oriented programming (AOP) [9] solve the issue of explicitly naming a class to extend. However, AOP is missing combination mechanism taking into account advices of other applicable aspects.

The automatic mitigation of interface incompatibilities is a general topic of service-oriented architectures [10, 13]. The call-by-meaning approach addresses the challenge for programming languages through a solving algorithm at the heart of any message dispatch. However, it provides no explicit abstraction mechanism [15].

## 5 EVALUATION AND FUTURE WORK

Implied methods should help with reusing method definitions in unanticipated situations. Thus, we plan to evaluate the approach through a case study. We will implement an application for managing personal contacts. As input it gets increasingly heterogeneous data objects representing contact information (for example vcard objects from various sources). After each increase in heterogeneity we adjust the application to work with the new objects once using ordinary object-oriented mechanisms and once using implied methods. We will evaluate the number of lines of code necessary to adjust the application and the number of reused methods [6].

One central issue of our approach is the introduction of ambiguity into the system description as multiple implied methods with the same name might be applicable based on the same interface. Thus, a remaining challenge are mechanisms and tool support to enable developers to limit the set of useful implied methods to be applied already during development time.

## ACKNOWLEDGMENTS

I gratefully acknowledge the financial support of the Research School of the Hasso Plattner Institute and the Hasso Plattner Design Thinking Research Program. I would also like to thank Stefan Ramson, Jens Lincke, and Robert Hirschfeld for countless fruitful discussions.

## REFERENCES

[1] Ruby documentation. Technical Documentation, 2017. URL https://ruby-lang.org/. Accessed 23th of February 2017.
[2] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications (ECOOP) 1990*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM. ISBN 0-89791-411-2. doi: 10.1145/97945.97982. URL http://doi.acm.org/10.1145/97945.97982.
[3] C. Clifton, T. D. Millstein, G. T. Leavens, and C. Chambers. Multijava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006. doi: 10.1145/1133651.1133655. URL http://doi.acm.org/10.1145/1133651.1133655.
[4] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–338, 2006. doi: 10.1145/1119479.1119483. URL http://dl.acm.org/citation.cfm?id=1119483.

[5] M. Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010. https://racket-lang.org/tr1/.

[6] W. B. Frakes and C. Terry. Software reuse: Metrics and models. *ACM Computing Surveys*, 28(2):415–435, 1996. doi: 10.1145/234528.234531. URL http://doi.acm.org/10.1145/234528.234531.

[7] R. P. Gabriel. The structure of a programming language revolution. In *Proceedings of the ACM Symposium on New Ideas in Programming and Reflections on Software (Onward!) 2012, part of SPLASH ’12, Tucson, AZ, USA, October 21-26, 2012*, pages 195–214, 2012. doi: 10.1145/2384592.2384611. URL http://doi.acm.org/10.1145/2384592.2384611.

[8] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. C. Kay. Back to the future: The story of squeak - A usable smalltalk written in itself. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA) 1997*, pages 318–326, October 1997. doi: 10.1145/263698.263754. URL http://doi.acm.org/10.1145/263698.263754.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of European Conference on Object-oriented Programming (ECOOP) 1997*, pages 220–242, 1997.

[10] S. A. McIlraith, T. C. Son, and H. Zeng. Semantic web services. *IEEE intelligent systems*, 16(2):46–53, 2001.

[11] Microsoft Corporation. C language specification 5.0. Technical report, 2012.

[12] D. A. Moon. Object-oriented programming with flavors. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA) 1986*, OOPSLA ’86, pages 1–8, New York, NY, USA, 1986. ACM. ISBN 0-89791-204-7. doi: 10.1145/28697.28698. URL http://doi.acm.org/10.1145/28697.28698.

[13] L. D. Ngan and R. Kanagasabai. Semantic web service discovery: State-of-the-art and research challenges. *Personal Ubiquitous Comput.*, 17(8):1741–1752, Dec. 2013. ISSN 1617-4909. doi: 10.1007/s00779-012-0609-z. URL http://dx.doi.org/10.1007/s00779-012-0609-z.

[14] J. Ortiz and D. Hall. Implicit classes. Scala Improvement Proposal, 2009. URL http://docs.scala-lang.org/sips/completed/implicit-classes.html. Accessed 23th of February 2017.

[15] H. Samimi, C. Deaton, Y. Ohshima, A. Warth, and T. D. Millstein. Call by meaning. In *Proceedings of the Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward!) 2014*, pages 11–28, October 2014. doi: 10.1145/2661136.2661152. URL http://doi.acm.org/10.1145/2661136.2661152.

[16] The Go Authors. Go documentation. Technical Documentation, 2017. URL https://golang.org/. Accessed 23th of February 2017.