# Proposal for an Object-Oriented Multiple Dispatch Mechanism

Miguel Oliveira e Silva
DETI-IEETA, University of Aveiro, Portugal
mos@ua.pt

## Abstract

Although multiple dispatch is recognized to be a very useful tool to properly solve difficult programming problems such as those resulting from binary methods, the large majority of existing object-oriented programming languages still don't support it. The few ones that provide such a mechanism do so in ways that either lie outside object-oriented modular construct (class), or possess some other limitations. In this short paper a new object-oriented language mechanism for multiple dispatch is presented (using Java as the base language) that is completely integrated within existing OO language constructs, and provides a simple, expressive, and generic programming dispatch tool.

***Categories and Subject Descriptors*** D.3.1 [*Programming Languages*]: Formal Definitions and Theory – semantics; D.3.2 [*Programming Languages*]: Language Classifications – object-oriented languages; D.3.3 [*Programming Languages*]: Language Constructs and Features abstract data types, control structures, inheritance, patterns, polymorphism, procedures, functions, and subroutines; D.3.m [*Miscellaneous*]: multiple dispatch, multimethods; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs

***Keywords*** multiple dispatch, object-oriented programming languages, multimethods, binary methods, type covariance, encapsulation, modularity, static typechecking, subtyping, inheritance, Java

## 1. Introduction

One of the distinctive characteristics of object-oriented programming languages is the existence of a (single) dynamic dispatch mechanism: the method to be executed depends on the runtime type of the object and not on the type of entity (variable, attribute, method argument) through which it's being used. This language construct is built using the inheritance mechanism and provides support for subtype polymorphism. This mechanism allows the construction of more modular and abstract programs, in which the usage of a type (in any of the language's typed entities) can be abstracted away from the concrete types of the objects being executed.

However, it is well known that this mechanism fails when the need arises to support the runtime selection of multiple types, as happens, for example, when binary methods are involved [2]. Its expressivity is also limited by the restriction that the method dynamic selection is hardwired with the object type used, so it is not easy to dynamically select a method that is not defined in the object's type that was used in the dynamic dispatch.

It is interesting to note that, in statically typed languages, single dynamic dispatch mechanisms are implemented with an implicit type safe covariant type change mechanism (subtyping), and that explicit covariant type change can only by safely implemented on non-mutable entity types (e.g. method result type). In mutable type entities, such as attributes or method arguments, a covariant type change is generally unsafe.

In an attempt to properly support multiple dispatch, multi-method languages were developed [4, 8] (other related approaches can also be found [1, 5]). In this approach, languages allow the definition of global methods (outside object classes) which are selected for execution depending on the runtime type of the objects passed as arguments. However, it can be argued that this mechanism goes against object-oriented modular architecture precisely because methods are defined externally to objects. It has also been acknowledged that this approach raises encapsulation and modularity problems [3, page 102]. In Tuple [7] some of these problems were solved, but at the cost of defining a new type construct (tuple classes), which becomes the destiny of dispatch messages (method invocation), replacing objects in that role (and losing some important features such as attributes and inheritance).

In this short paper a new object-oriented language mechanism for multiple dispatch is presented. To help the presentation, we will use a Java language extension (named MD-Java[1]) and the problem of the collision of 2D figures (rectangles, circles, etc.) with be used as an example.

## 2. The Proposal

We need to conciliate two apparently conflicting mechanisms: the necessity for dynamic dispatch on a tuple of object types, and to find a target object where such methods are to be defined and executed (hence ensuring objects are the main modular language structure). Since making one of the objects of the tuple the target of such selection in not a good solution, we had the idea of generalizing the object creation mechanism for that goal. Usually, if creation patterns [6] (such as factory methods) are not used, there is only one place in object-oriented programs in which the type of an object is known: when it is created. In our proposal that restriction is eliminated.

### 2.1 Dynamic Creation Classes

Since a language dispatch mechanism is required, dynamically created classes should be syntactical different from standard classes. In our approach, a language construct similar to methods argument definition was used.

[1] Multiple Dispatch Java

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

*CONF 'yy*, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

**Listing 1.** Figure classes

```
public class Figure {...}

public class Rectangle extends Figure {...}

public class Circle extends Figure {...}
```

Listing 1 shows some classic figure classes (deprived of their interface). A class abstracting the interaction of two figures (with a minimal interface) is presented in listing 2 and classes implementing the interaction some concrete figures are presented in listing 3.

**Listing 2.** Two figure multiple dispatch class

```
public abstract class TwoFigure(Figure f1,Figure f2) {
   public boolean collide() {
      boolean result = (distance() <= 0);
      if (result)   collisionsDetected++;
      return result;
   }

   public abstract double distance();

   protected int collisionsDetected = 0;
}
```

**Listing 3.** Example of concrete dispatch classes

```
public class TwoCircle(Circle c1,Circle c2)
      extends TwoFigure {
   public double distance() {
      return Math.sqrt((c1.x()-c2.x())*(c1.x()-c2.x()) +
                       (c1.y()-c2.y())*(c1.y()-c2.y())) -
             c1.radius() - c2.radius();
   }
}

public class CircleRectangle(Circle c,Rectangle r)
      extends TwoFigure {
   public double distance() {
      ...
   }
}
```

Syntactically the only difference between dynamic creation classes and normal classes is the existence of class arguments. The inheritance relation of such classes retain normal inheritance semantics, enhanced by the requirement of conformance[2] between the class arguments (`Circle` is a subtype of `Figure`). Within the class, the formal class arguments (`f1` and `f2` in class `Figure`) are semantically similar to to the object auto-reference `this` (in particular, its value is immutable in all the objects lifetime).

### 2.2 Dynamic Creation Instruction

Listing 4 exemplifies the use of the new creation instruction (line 10).

**Listing 4.** Multiple dispatch example

```
1  // import all dispatch classes in interaction package:
2  import dispatch interaction.*;
3
4  public class Test {
5     public static void main(String[] args) {
6        Figure f1 = new Circle(10,10,5);
7        Figure f2 = new Rectangle(10,12,20,15);
8        ...
9        // Multiple dispatch on all TwoFigure descendants
10       TwoFigure tfi = new (f1,f2).TwoFigure();
11       // A CircleRectangle object was created!
12       System.out.println("distance = "+tfi.distance());
13    }
14 }
```

[2] Non-variant or covariant type change.

A logical consequence of this mechanism is the necessity to impose some extra semantic rules to the language (so that the compiler can fulfill its job). The first one is the obligation that all multiple dispatch class combinations of existing non-abstract (instantiable) classes (`Circle`, `Rectangle`, etc.) should match a conforming non-abstract multiple dispatch class (or else, a new kind of type hole would arise). Another convenient rule is the irrelevance on the order of object arguments in the dynamic creation instruction (in the example, a `CircleRectangle` object would be also created with instruction "`new (f2,f1).TwoFigure()`"). This creation dispatch instruction will instantiate the closest combination of the dispatch objects types involved.

With the exception of the moment of creation, multiple dispatch objects are absolutely similar to normal objects, and all other object oriented constructs (definition of attributes, generic types, classic OO polymorphism, etc.) can be used. It should also be noted that this approach to multiple dispatch does not compromise encapsulation nor modularity.

## 3. Final Remarks

Some details on the semantics of the proposed language mechanism were not presented due to the limitation in the paper size (in particular, nothing was mentioned regarding the single inheritance limitation of `Java`). Nevertheless, it should be mentioned that the author stands for the type safety of the mechanism proposed, and is actively working on a prototype compiler for this mechanism.

Finally, an interesting positive side-effect of this mechanism is that it not only provides a simple language solution to *visit* and *creation* design patterns [6], but it also promotes a new programming tool based on external object dispatching (definition of classes that dispatch on external, possibly unrelated, object types), generalizing the object-oriented dispatch mechanism.

## References


[1] J. Boyland and G. Castagna. Parasitic methods: an implementation of multi-methods for Java. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 66–76. ACM Press, 1997. ISBN 0-89791-908-4. .

[2] K. Bruce, L. Cardelli, G. Castagna, G. T. Leavens, and B. Pierce. On Binary Methods. *Theor. Pract. Object Syst.*, 1(3):221–242, Dec. 1995. ISSN 1074-3227. URL `http://dl.acm.org/citation.cfm?id=230849.230854`.

[3] K. B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, Cambridge, MA, USA, 2002. ISBN 0-262-02523-X.

[4] C. Chambers. Object-Oriented Multi-Methods in Cecil. In O. L. Madsen, editor, *Proceedings ECOOP'92, LNCS 615*, pages 33–56, Utrecht, The Netherlands, Jun 1992. Springer-Verlag.

[5] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145. ACM Press, 2000. ISBN 1-58113-200-X. .

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[7] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on Tuples. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 374–387. ACM Press, 1998. ISBN 1-58113-005-8. .

[8] G. L. Steele, Jr. *Common LISP: The Language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.