Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

External Dispatch: Yet Another Object-Oriented Single and Multiple Dispatch Mechanism

Miguel Oliveira e Silva[a]

a. DETI-IEETA, University of Aveiro, Portugal

Abstract Although multiple dispatch properly solves difficult programming problems such as binary methods, the large majority of existing object-oriented programming languages still don't support it. The few languages that provide such a mechanism do so in ways that either lie outside an object-oriented modular construct (class), or possess some other limitations, such as arbitrarily choosing one of the dispatch object types as the repository for dispatch methods. This article presents a new object-oriented language mechanism for single and multiple dispatch. As a proof of concept, we will use `Java` as the base language. This mechanism is compatible with existing object-oriented language constructs, and provides a simple, expressive, and universal dispatch tool. Our proposal introduces a new object-oriented external dispatch mechanism, which complements the traditional object-oriented internal single dispatch mechanism. This new mechanism is applicable not only to multiple dispatch, but can also be used as an alternative to decorators and some creational design patterns. A more complete realization for external dispatch motivated the definition of a new singleton language mechanism that is also presented.

Keywords multiple dispatch; single dispatch; external dispatch; object-oriented programming languages; multimethods; binary methods; type covariance; encapsulation; modularity; static typechecking; subtyping; inheritance; mixins, decorator design pattern; creational design patterns; singleton; Java.

1 Introduction

A distinctive characteristic of object-oriented programming languages is the single dynamic dispatch mechanism: the method to be executed depends on the runtime type of the target object. This language construct is built using the inheritance mechanism and provides support for subtype polymorphism. It allows the construction of more modular and abstract programs. We consider this dispatch as internal, because the

Miguel Oliveira e Silva. *External Dispatch: Yet Another Object-Oriented Single and Multiple Dispatch Mechanism*. In Journal of Object Technology, vol. 16, no. 2, 2017, pages 1:1–18. doi:10.5381/jot.2017.16.2.a1

target object is also the dispatch object, i.e. the method to be executed is defined in the type of the dispatch object.

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 [BCC+95]. Its expressivity is limited also 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 class.

In an attempt to properly support multiple dispatch, multi-method languages were developed [Ste90, Cha92]. Multi-method languages allow the definition of global methods (outside object classes) that are selected for execution depending on the runtime types 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 classes. It has also been acknowledged that this approach raises encapsulation and modularity problems [Bru02, page 102]. In Tuple [LM98] some of these problems were solved, but at the cost of defining a new type construct (tuple classes), that becomes the receiver of dispatch messages (method invocation), replacing objects in that role (and losing some important features such as fields and inheritance). Other approaches can also be found in [BC97, CLCM00, KRLS13].

This article proposes a new object-oriented language mechanism for single and multiple dispatch. As a proof of concept, we developed a Java language extension that implements this mechanism.[1] To help the presentation, the problem of the intersection of 2D shapes (rectangles, circles, etc.) will be used as an example.

The contributions of this article are: (1) a new object-oriented single and multiple dispatch mechanism; (2) the definition of a set of rules to enforce its static safety; (3) a new language mechanism for singletons.

This article is organized as follows. Section 2 discusses the motivation for this work. Section 3 presents some important existing approaches to support multiple dispatch. Our proposal is presented in section 4. The singleton mechanism is presented in section 5. Some single dispatch applications of the mechanism are presented in section 6. Finally, some concluding remarks are made in section 7.

2 Motivation

Object-oriented programming promotes a modular methodology in which objects can be viewed as self-contained entities that expose an abstract behavior and hide the implementation from their external clients. Toward such goal, object-oriented languages provide mechanisms for encapsulation, and mechanisms for subtype polymorphism and dynamic dispatch. These mechanisms enhance the support for a modular abstract definition and use of objects. A client only needs to know the object's interface and what it does, and not how it is implemented. Also, different types of objects can be used through the same polymorphic typed entity, making the client's code functional even with yet to be developed object types.

For instance, developing a graphical application involving operations such as drawing shapes like rectangles and circles, an object-oriented approach may define the abstract common behavior of shapes, as well as non-abstract implementations for each type of shape (figure 1). The program in listing 1 uses encapsulation and dynamic dispatch to solve the problem of drawing all shapes in an array.

[1] Named ED-Java (External Dispatch Java).

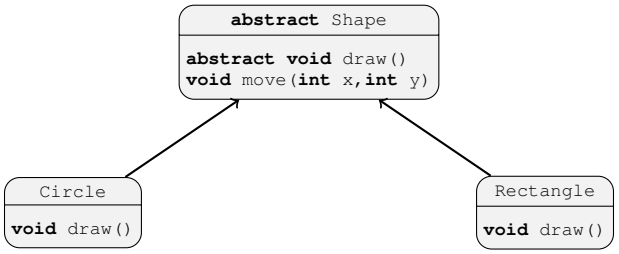


Figure 1 – Shape classes for a graphical application.

```
public void drawAll(Shape[] shapes) {
  for(Shape s: shapes)
    s.draw();
}
```

Listing 1 – Draw shapes in array

The simplicity and abstraction of the solution depends on the fact that a proper implementation of the redefined method (`draw` in the example given) depends only on its enclosing, shape subtype, class. Hence, to dynamically select the proper method, the object's type suffices. Object-oriented programming is well adapted to such internal single dispatch patterns.

More serious problems arise if the desired method depends on more than one type, such as in binary methods [BCC+95]. For example, in a graphical application, an operation to test the intersection of two shapes requires the knowledge of both shape types. Detecting the intersection of two circles is trivial, but the intersection of a circle with a rectangle requires a different, more complex, algorithm. If both shapes are handled through type `Shape` entities (such as in listing 1), then other than being a `Shape`, there is no static knowledge on what are the types of the objects involved, and a dynamic dispatch approach is required. Unlike the usual object-oriented internal single dispatch mechanism, to properly solve this problem we need a dispatch algorithm involving two object types; i.e. a multiple dispatch mechanism.

3 Representative existing approaches

3.1 Global multi-methods

One possible approach is simply to *remove methods from classes*, and allow dynamic dispatch on argument types of global functions (listing 2). That is the solution taken by `CommonLoops` [BKK+86], `CLOS` [Kee89], `CECIL` [Cha92].

```
boolean intersect(Circle c1, Circle c2) { ··· }
boolean intersect(Circle c, Rectangle r) { ··· }
...
```

Listing 2 – Global multi-methods

```
public abstract class Shape {
  public abstract boolean intersect(Shape other);
}

public class Circle extends Shape {
  public boolean intersect(Shape@Circle other) { ··· }
  public boolean intersect(Shape@Rectangle other) { ··· }
}

...
```

Listing 3 – Class multi-methods written in MultiJava

```
tuple class (Circle c1, Circle c2) {
  boolean intersect() { ··· }
  int distance() { ··· }
}

tuple class (Circle c, Rectangle r) {
  boolean intersect() { ··· }
  int distance() { ··· }
}

...
```

Listing 4 – Tuple multi-methods

This approach is not very object-oriented, because these multi-dispatch methods are not part of a class/object.

3.2 Class multi-methods

Another possibility is to put methods in the *class of one of the dispatch objects*. This is the approach taken by Parasitic Methods [BC97], MultiJava [CLCM00], and Featherweight Multi Java [BCV07]. Listing 3 shows an example written in MultiJava.

A problem with this approach is that it can represent an overspecification of a class, because the class might contain a behavior that is also strongly related with another class. Also, such mixed behavior might reduce the class modularity as it might strengthen the coupling with other classes (in the example given, knowledge of rectangles is required within the circle class). As a result, such a class needs to explicitly support a behavior that should probably be elsewhere.

Another potential problem is the asymmetric treatment that is given to dispatch objects, that occurs when one is selected as the dispatch target.

Finally, this approach compromises the "open-closed principle" [Mey97]. If a new intervening dispatch type is added to the program (an ellipse, for example), this approach requires that existing classes need to be opened and extended with new dispatch methods.

The open-class approach [CLCM00] alleviates this last problem, and other approaches exist for symmetric dispatch [CLCM00, Cha92], but neither of them solves the class's *overspecification* problem.

3.3 Tuple multi-methods

Another possibility is to define a new structuring module – a *tuple* – in which all these methods are declared and implemented (listing 4), as taken in Tuple [LM98]. The problem with this approach is that tuple multi-methods are not part of an object. Hence, important features are lost, such as fields, explicit subtyping and inheritance, and abstract methods.

4 External dispatch proposal

It is necessary to reconcile two apparently conflicting requirements: dynamic dispatch using a tuple of objects, and find a target object where dispatched methods are defined and executed (hence ensuring that objects and classes remain the essential modular language structures). Existing approaches either do not select a target object (sections 3.1 and 3.3), or select one of the tuple's dispatch objects (section 3.2). However, not only is choosing one of the dispatch objects as target arbitrary (which one?), but, more important, its class might not be the correct location to place this joint behavior. In the example given, the intersection behavior of two shapes is not specific of a single shape type, but to both shape types. Additionally, if a new behavior specific to intersection shapes, for example define a method to count the number of collisions, is added to the program, then it makes little sense to place it inside a circle or any other single shape class. The joint behavior of two or more objects, as expressed in multiple dispatch applications, should be a type abstraction of its own. A *circle-rectangle* intersection, should be placed in a joint *circle-rectangle* class, not in the class of either of these two shapes.

This view raises the problem of how to find the proper target object. In the class multi-methods approach, this problem has a simple solution: the object is one of the dispatch objects, but if the joint behavior is a separate abstraction then it is necessary to select an object *external* to the set of dispatch objects. The solution proposed is to reuse and generalize the *object creation mechanism*. The idea is to perform the dynamic dispatch not directly in method invocation, but in object creation. Hence, a dynamic dispatch of a method is separated in two phases: (1) instantiate the correct dispatch target object, and then (2) invoke the method with the common object-oriented internal dispatch semantics.

In existing object-oriented languages such as `Java`, the creation expression unambiguously identifies the object's class. In our proposal, the creation expression is extended to (optionally) include an object tuple as a prefix argument (see section 4.2) so that it unambiguously identifies the object's *type*, but not its *non-abstract class*. It is up to the semantic rules of the mechanism to select the object's class using a dynamic external dispatch on the tuple.

When designing this new mechanism, special care was taken so that the syntactic and semantic use of original language constructs remain the same (meaning that existing programs in the base language are valid and work as before). Also, we tried to take advantage of possible interesting synergic behaviors when the new extensions are used with other existing language constructs (abstract classes and methods, parametric polymorphism, inheritance, etc.).

Three major extensions to the base language (`Java` was used as a case study) are proposed:

1. an extension to class declaration named external dispatch classes (section 4.1);

```
normalClassDeclaration:
    classModifier* 'class' Identifier typeParameters? externalDispatchArguments?
    superclass? superinterfaces? classBody
  ;

externalDispatchArguments:
    '(' formalParameterList ')' |
    '(' '{' formalParameterList '}' ')'
  ;
```

Listing 5 – Class declaration grammar

2. an extension to the creation expression (section 4.2);

3. a mechanism for singletons (section 5).

First we will informally present the mechanism, leaving a more formal presentation to section 4.5.

4.1 External dispatch classes

Considering Java's syntax description in Java Language Specification [GJS+14, page 193], listing 5 presents the new class declaration grammar with additions highlighted (`Java8` grammar expressed in `ANTLR`).

External dispatch classes are *classes with dispatch arguments*.[2] For example:

```
public class A<T>(B b,C c) extends D implements E { ··· }
```

An external dispatch generic class `A` is defined, with two dispatch arguments of type `B` and `C`, extending a class `D` and implementing an interface `E`. The application of the new extended creation expression (defined in the next section) to class `A` requires passing a dispatch object tuple (o_1, o_2), of type (T_1, T_2), conforming to the formal parameter list of the external dispatch arguments, i.e., $T_1 <: B \ \wedge \ T_2 <: C$, where $<:$ represents a subtype relation.

Alternatively, the class declaration might prescribe an unordered list of dispatch arguments:

```
public class UA<T>({B b,C c}) extends D implements E { ··· }
```

This class dispatches to both: $(T_1 <: B \ \wedge \ T_2 <: C) \ \vee \ (T_1 <: C \ \wedge \ T_2 <: B)$.

4.2 External dispatch creation expression

The grammar of the extended creation expression (based on [GJS+14, pages 482-483]) is shown in listing 6.

The creation expression is extended with an object tuple prefix (object dispatch tuple). For example:

```
B b; C c;
...
A<Integer> a = new (b,c).A<>(); // extended creation expression
...
```

[2] Classes in Scala [OMM+04] also have arguments but the semantics is very different.

```
unqualifiedClassInstanceCreationExpression:
    'new' classOrInterfaceTypeToInstantiate '(' argumentList? ')' classBody?
  ;

classOrInterfaceTypeToInstantiate:
    annotation* dispatchTuple? identifier ('.' annotation* Identifier)* typeArgumentsOrDiamond?
  ;

dispatchTuple:
    '(' argumentList ')' '.'
  ;
```

Listing 6 – Creation expression grammar

```
ImportDeclaration:
    SingleTypeImportDeclaration |
    TypeImportOnDemandDeclaration |
    SingleStaticImportDeclaration |
    StaticImportOnDemandDeclaration |
    DispatchImportDeclaration
  ;

DispatchImportDeclaration:
    'import' 'dispatch' ( '*' | TypeName | PackageOrTypeName '.' '*' ) ';'
  ;
```

Listing 7 – Import dispatch grammar

If successfully compiled and run, this expression creates an object with type `T<Integer>` subtype of `A<Integer>`, and `T<>` is the best conforming type match for the object types in the dispatch tuple (b, c). Rule 8 in page 12 presents a more formal definition for this criterion. Static types of the entities in the object dispatch tuple, must conform to the types of the formal dispatch arguments of the external dispatch class (rule 9 in page 12).

External dispatch object creation explicitly refers the type of an external dispatch class (in the example: `A<>`), but, unlike normal object creation, it can instantiate an object of a subtype `T<>` of `A<>`. The set of possible classes `T<>` from which the "best conforming type" is chosen is fixed in compile-time. Also, the set of types that might be involved in the dispatch argument tuple (in the example, all subtypes of `B` and `C`) may not explicitly appear in the class where the external dispatch object is created. The matching from all valid combinations of dispatch arguments types to a non-abstract target external dispatch class is also chosen and fixed at compile-time. Because some the classes involved might not be explicitly referred in the external dispatch creation expression, the programmer indicates which classes to consider using a new kind of dispatch import statement (listing 7).

This new statement allows the programmer to explicitly select the list of classes to be considered for external dispatch. It includes both external dispatch classes, and the classes used in dispatch arguments. A new `import dispatch *` statement was added to explicitly include all classes of current package. For example:

```
import dispatch *;                  // all classes in current package
import dispatch p.B;                // class B in package p
import dispatch java.lang.*;        // all classes in package java.lang
```

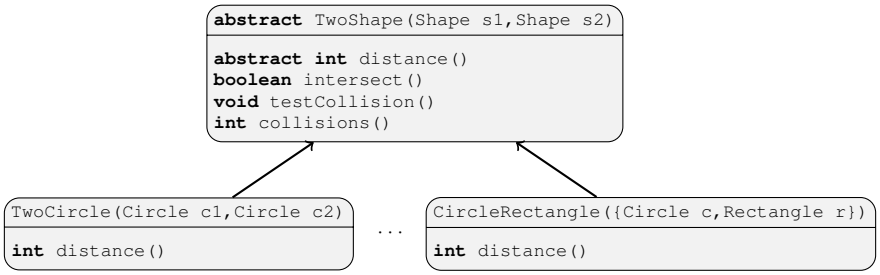


Figure 2 – External dispatch shape classes.

```
public abstract class TwoShape(Shape s1,Shape s2) {
  public abstract double distance();

  public boolean intersect() {
    return distance() <= 0;
  }

  public void testCollision() {
    if (intersect())
      collisionsDetected++;
  }

  public int collisions() {
    return collisionsDetected;
  }

  protected int collisionsDetected = 0;
}
```

Listing 8 – Two shape multiple dispatch class

In short, import dispatch statement creates a dependency between the compilation unit (i.e. the whole file) and the classes or packages expressed. A change (class added or removed) will require its recompilation.

4.3 Shape example

Figure 2 and listings 8 and 9 exemplify possible external dispatch classes for a shape subtype graph presented in figure 1. Class `CircleRectangle` can dispatch a circle-rectangle or a rectangle-circle tuple. The external dispatch object creation expression will ensure that the external dispatch class created will be the best match (discussed ahead) to the dispatch object tuple.

Listing 10 exemplifies the use of the external dispatch creation expression for the shapes example (for the exemplified shapes, a successful compilation requires also a `RectangleRectangle` external dispatch class).

The inheritance relation of external dispatch classes retains normal inheritance semantics, with the addition requirement of conformance in homologous dispatch arguments. Conformance means a non-variant or covariant type change of formal dispatch arguments in the same position (note that this semantics is not changed by an unordered dispatch argument declaration).

To ensure that inherited methods work as expected in subclasses, an automatic renaming of formal dispatch arguments occurs when external dispatch classes are

```
public class TwoCircle(Circle c1, Circle c2) extends TwoShape {
  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 TwoShape {
  public double distance() {
    ...
  }
}

... // other TwoShape external dispatch classes
```

Listing 9 – Example of non-abstract dispatch classes

```
// consider all classes from default package for dispatch:
import dispatch *;

public class Test {
  public static void main(String[] args) {
    Shape s1 = new Rectangle(10,12,20,15);
    Shape s2 = new Circle(10,10,5);
    ...
    // Multiple dispatch on all TwoShape descendants
    TwoShape tsi = new (s1,s2).TwoShape();
    // A CircleRectangle object was created!
    System.out.println("distance = "+tsi.distance());
  }
}
```

Listing 10 – Multiple dispatch example

inherited.[3] This renaming applies to homologous formal dispatch arguments. In the example given, formal dispatch arguments `s1` and `s2` from class `TwoShape`, and formal dispatch arguments `c1` and `c2` from class `TwoCircle` refer to the same objects. Hence, all methods implemented in `TwoShape` behaves as expected within the context of a `TwoCircle` object.

Another restriction that applies to external dispatch classes is the "inheritance" of constructor methods. An external dispatch class `Y` that extends another external dispatch class `X` containing constructors with arguments is required to override those constructors, otherwise, a "method not found" error could arise at runtime. The exception is the default constructor that is always defined in classes without constructor declarations.

4.4 Modularity

The external dispatch mechanism is modular because the addition of a new class that is a subclass of a dispatch argument (for instance, a square or a polygon shape) has no impact to existing classes. Eventually, new external dispatch classes might be necessary to ensure a complete instantiation (see rule 7 in page 11).

[3] This behavior differs from `Java` normal semantics.

4.5 Static rules

To enforce the static safety of the external dispatch mechanism, the compiler verifies the rules presented in this section.

Consider the following external dispatch classes: class C with $n \geq 1$ dispatch arguments $c_1, \ldots, c_n$, of types $C_1, \ldots, C_n$; class D with $m \geq 1$ dispatch arguments $d_1, \ldots, d_m$, of types $D_1, \ldots, D_m$; and class E with $l \geq 1$ dispatch arguments $e_1, \ldots, e_l$, of types $E_1, \ldots, E_l$.[4]

We use the notation $X \trianglelefteq Y$ to express that X is directly, or indirectly, a subclass of Y (note also that $X \trianglelefteq X$). Also, we define the set $\text{inst}(A)$ of all non-abstract classes that are subclasses of A:

$$\text{inst}(A) = \{X \mid X \trianglelefteq A \ \wedge \ X \text{ is not abstract}\}.$$

Finally, we define the set $\text{inst}(C, i)$ applicable to the external dispatch class C and to integer i, that contains all the non-abstract classes that can be used as the ith dispatch argument ($1 \leq i \leq n$). This set contains $\text{inst}(C_i)$ and, for all subclasses D of C that have unordered dispatch arguments, also $\{X \mid \forall i \in \{1, \ldots, n\}, X = D_i\}$.

4.5.1 External dispatch classes rules

External dispatch classes behave like normal classes except for the restrictions imposed by the rules presented in this section.

Rule 1 ***Conforming dispatch arguments****: The subclass relation between external dispatch classes requires conforming dispatch arguments.*

$$\forall D, C : D \trianglelefteq C \implies (m = n) \ \wedge \ (\forall i \in \{1, \ldots, n\}, \ D_i <: C_i)$$

This rule is required to ensure that the subclass relation between two external dispatch classes represents also a safe subtype relation; i.e. any instance of a subclass D can be used in expressions of type C.

Rule 2 ***Non-conflicting subclasses 1****: Different external dispatch classes that are subclasses of a common external dispatch class, must differ in at least one dispatch argument type.*

$$\forall C, D, E : \ D \trianglelefteq C \ \wedge \ E \trianglelefteq C \ \wedge \ D \neq E \implies \exists i \in \{1, \ldots, n\}, \ D_i \neq E_i$$

Note that because of rule 1, $m = l = n$.

Rule 3 ***Non-conflicting subclasses 2****: For an external dispatch class with unordered dispatch arguments, the verification of the previous rule must apply to all permutations of dispatch arguments.*

If E has unordered dispatch arguments, and P_j is one of the possible $n!$ permutations of numbers $1, \ldots, n$, then:

$$\begin{aligned} \forall C, D, E : \quad & D \trianglelefteq C \ \wedge \ E \trianglelefteq C \ \wedge \ D \neq E \implies \\ & \forall j \in \{1, \ldots, n!\} \ \exists i \in \{1, \ldots, n\}, \ D_i \neq E_{P_j(i)} \end{aligned}$$

In short, for the external dispatch to work, it is necessary that a class is unambiguously identifiable within the set of all subclasses of C.

[4] The place of dispatch arguments is the declared one, regardless of being an ordered or unordered list.

Rule 4 ***Class formal dispatch arguments with final semantics****: Restrictions applicable to keyword* `this` *([GJS+14, page 480]) are extended to all formal dispatch arguments (in class C the restrictions apply to formals: $c_1, \ldots, c_n$).*

The binding to formal dispatch arguments occurs when the object is created (as happens with `this` keyword), and remains the same while the object is alive.

Rule 5 ***Overriding of constructors****: All subclasses of an external dispatch class C, must implement constructors with argument type lists equal to those in the constructors of C except when class C has no declared constructors.*

Since the external creation expression instantiates a subclass of C, all possible subclasses must implement at least the constructors present in C, otherwise a creation error could arise at runtime.

Rule 6 ***Formal dispatch arguments renaming****: The code of an external dispatch class is inherited as if its formal dispatch arguments are the homologous dispatch arguments of the subclass.*

$$\forall D, C : D \trianglelefteq C \ \wedge \ D \neq C \implies \forall\, i \in \{1, \ldots, n\}\ c_i = d_i \ \wedge \ c_i \textit{ is hidden in } D$$

This means two things. First, the superclass formals cannot be used in the context of the subclass (they were renamed). Second, a subclass is assured that inherited methods behave as if the class's dispatch arguments were used in place of the homologous superclass dispatch arguments. In the presented shape interaction example, if formals `s1` and `s2` were used in a method, then this method would word as expected in subclasses of `TwoShape` as if the subclasses dispatch arguments formals were in that superclass.

Rule 7 ***Complete instantiation****: The set of all non-abstract subclasses of any external dispatch class, should be enough to ensure the dispatch of all possible dispatch tuple non-abstract types.*

Formally, considering a tuple $T = (R_1, \ldots, R_n)$ such that $R_i \in inst(C, i)$, and a non-abstract external dispatch class D. Then, for an external dispatch class C:

$$\forall R_1 \in inst(C,1)\ \ldots\ \forall R_n \in inst(C,n)\ \exists D : \quad \begin{array}{l} D \in inst(C) \wedge \\ (R_1 <: D_1 \wedge \ldots \wedge R_n <: D_n) \end{array}$$

If we define the set $\text{inst}(C, T)$, as the set of non-abstract subclasses of external dispatch class C that are possible targets of tuple T, then this rule states that for all possible tuples T, $\text{inst}(C, T)$ in not an empty set.

This rule serves the purpose of assuring that all possible valid combinations of the non-abstract types of the dispatch arguments of an external dispatch class, conform to a valid dispatch argument type tuple of at least one instantiable external dispatch subclass. A sufficient condition for this rule, is a non-abstract class C.

Type distance: Consider $\text{dist}(A, B)$ to be an integer function that returns the distance between the class A and the class B. The distance between two classes is the minimum number of edges in the subclass directed graph from A to B. The subclass directed graph is defined by considering classes to be its nodes, and the subclass relation its edges (directed from the subclass to the parent class). This function is not defined if no path exists from A to B (i.e. if A is not a subclass of B); and is zero if $A = B$. In the subtype graph in figure 3, the distance between types `Square` and `Shape` is two, and $\text{dist}(\texttt{Rectangle}, \texttt{Circle})$ is not defined.

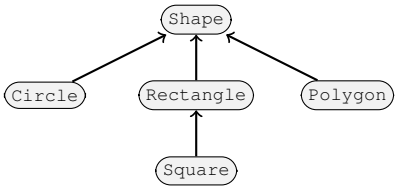


Figure 3 – Subclass directed graph for some shapes.

Rule 8 ***Unambiguous instantiation****: The set of all non-abstract subclasses of an external dispatch class C should be no less, and no more, than the necessary to ensure that for inst(C,T), there is no ambiguity in the selection of a single best-match non-abstract target external dispatch class.*

Formally, considering a tuple T as defined in rule 7, the integer function:

$$\textit{dist-tuple}(T,C) = \sum_{i=1}^{n} \textit{dist}(R_i, C_i),$$

and function min(C,T) that contains the non-abstract subclasses of C with minimum distance to dispatch tuple T, i.e.:

$$\begin{aligned} \textit{min}(C,T) = \{M \mid \quad & M \in \textit{inst}(C) \wedge \\ & (\forall D : D \in \textit{inst}(C), \textit{dist-tuple}(T,M) \leq \textit{dist-tuple}(T,D))\} \end{aligned}$$

then, for all possible tuples T, min(C,T) is composed of a single target external dispatch class (named best match).

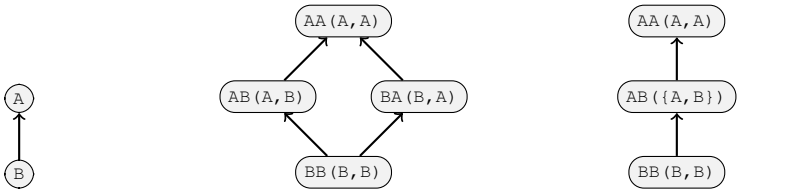
Single and multiple inheritance: The unambiguous instantiation rule is more difficult to ensure in languages that do not support multiple inheritance (such as `Java`). Consider the subtype graph in figure 4a. The graph in figure 4b is the result of all combinations of pairs of these two classes. In single inheritance languages such diamond shape inheritance graph is not possible to implement, so some of its natural subtype relations cannot be expressed. Clearly if all classes are not abstract, and class `BB` is not implemented, a compiler error occurs. A possible solution is to implement `BB` arbitrarily extending one of the `AB`, or `BA` classes. A better solution is to use a single external dispatch class with unordered dispatch arguments `AB` (figure 4c).

4.5.2 Dispatch creation expression rules

Consider the following dispatch tuple: $X = (x_1, \ldots, x_k)$, in which x_i are valid expressions of type X_i, and the creation expression: new$(x_1, \ldots, x_k).C$(OptArgumentList). Functions ord(C) and unord(C) indicate if the external dispatch class C has ordered or unordered dispatch arguments. Consider also permutation function P defined in rule 3.

Rule 9 ***Conforming dispatch tuple****: In an external dispatch creation expression, the dispatch tuple X must conform with the homologous types of the dispatch arguments of the external dispatch class C, or with a permutation of those arguments if C has unordered dispatch arguments.*

$$\begin{aligned} & (k = n) \ \wedge \ (\textit{ord}(C) \implies (\forall i \in \{1,\ldots,n\},\ X_i <: C_i)) \wedge \\ & (\textit{unord}(C) \implies (\forall i \in \{1,\ldots,n\}\ \exists j \in \{1,\ldots,n!\},\ X_i <: C_{P_j(i)})) \end{aligned}$$



(a) Example subtype graph. (b) Ordered combinations. (c) Unordered combinations.

Figure 4 – Multiple inheritance problem: combination of classes.

Rule 10 ***Constructor rule****: Class C must contain a constructor applicable to OptArgumentList.*

Rule 11 ***No null references in dispatch tuple****: The dispatch tuble cannot contain a null reference.*

$$\forall i \in \{1, \ldots, n\},\ x_i \neq null$$

Rule 11 is the exception to all presented rules, because it is enforced only at runtime.

4.6 External dispatch classes *versus* normal classes

Since the behavior of external dispatch classes retains the usual semantics of normal classes (except on creation), we could consider normal classes a special case of a *zero* external dispatch class (there is no problem in allowing an empty dispatch tuple in the creation expression). However, in this article we consider external dispatch classes as classes with at least one dispatch argument.

4.7 Current implementation

For this proposal a compiler extending `Java8` is being developed in `ANTLR-v4` [Par13]. A proof of concept for automatic code generation was implemented using Java reflection library for object instantiation, and an associative array for implementing the dispatch matching (all possible dispatch combinations are mapped to non-abstract external dispatch classes).

5 Singletons

So far, the external dispatch mechanism creates a new object in each external dispatch creation expression. However, sometimes we would like to reuse the *same* target object in several dispatch operations. In particular, if there are fields declared within the external dispatch class, we might want to ensure that the same object is the target of several external dispatch operations.

On the other hand, the necessity for types with single instances also occurs in normal object-oriented programming. A design pattern that approaches such a goal is the *singleton* [GHJV95]. However, due to its implementation, the singleton design pattern ensures only that a *class* has a single instance, which is not necessarily the same thing as a single instance per type. The problem here is the negative interference with inheritance and generic class type language mechanisms. A dynamically dispatched singleton for each class in a subclass hierarchy is difficult to implement. In the case of

```
public singleton class TwoCircle(Circle c1, Circle c2) extends TwoShape {
  public double distance() {
    ...
  }
}

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

Listing 11 – Example of singleton dispatch classes

generic types, it is not taken into consideration that a generic class is a type constructor (i.e. different types for different generic type instantiations). For all these reasons a language supported mechanism for type singletons is desirable.

5.1 Our proposal

Syntactically, our proposal simply adds a new class modifier keyword: `singleton`. Semantically, a singleton class ensures a single object per type (instead of a single object per class). When applied to external dispatch classes, this mechanism ensures a singleton per dispatch tuple object combination. In short, it ensures a singleton for each combination of object self-reference `this` and, if any, all objects referred by the class's dispatch arguments formals.

The application of the creation expression to a singleton class always returns the same object reference (the instantiation occurs only in the first creation attempt). To avoid undesirable side-effects, singleton classes can only declare the argument-less constructor. Also, "dead" singleton objects cannot be garbage-collected (or else, multiple versions of a singleton could arise during program execution).[5]

The reason for this mechanism to select a single object tuple combination, instead of a single object type tuple combination, lies in the semantics of the formal dispatch argument entities: Formal dispatch arguments possess similar semantics to the `this` keyword (rule 4). If the singleton semantics were related to the tuple object *types*, formals binding to objects would only occur in the first creation attempt, resulting in an erroneous behavior when different objects of the same types was used in later external creation expressions. Hence, a saner behavior is to consider that the singleton targets not only a single `this` reference, but also a single object dispatch tuple.

In this article we will exemplify the mechanism in the context of external dispatch (though its semantics for normal classes is not hard to foresee).

Getting back to our shape intersection example, consider the convenience to count the number of collisions between two specific shapes. The usage of singleton external dispatch types ensures a correct count (see listing 11).[6]

6 Other applications

Multiple dispatch problems, such as the shape interaction example, are natural applications of the external dispatch mechanism (in fact, our proposal was the result

[5] A more complete semantics should also ensure thread-safe single instantiation.

[6] The TwoShape class is abstract, thus it is irrelevant to annotate it as a singleton.

```
public abstract class ExtendedShape(Shape s) extends Shape {
  public abstract boolean isConvex();

  @Override public void draw() {
    s.draw();
  }

  @Override public void move(int x, int y) {
    s.move(x,y);
  }
}
```

```
public class ExtendedCircle(Circle c) extends ExtendedShape {
  public boolean isConvex() {
    return true;
  }
}
```

```
public class ExtendedRectangle(Rectangle r) extends ExtendedShape {
  public boolean isConvex() { // applies also to Square
    return true;
  }
}
```

```
public class ExtendedPolygon(Polygon p) extends ExtendedShape {
  public boolean isConvex() {
    ··· // algorithm to verify the convexity of polygon p
  }
}
```

```
// necessary to observe complete instantiation rule (rule 7)
public class ExtendedExtendedShape(ExtendedShape es) extends ExtendedShape {
  public boolean isConvex() { // applies to all ExtendedShape
    return es.isConvex();
  }
}
```

```
public void doSomething(Shape[] shapes) { // extended shape usage:
  for(Shape s: shapes)
    if (new (s).ExtendedShape().isConvex()) { // external dispatch to proper isConvex
      ...
    }
}
```

Listing 12 – Extended shape example

of a long time quest for an object-oriented solution to such problems). However, this mechanism can give elegant solutions to other important applications.

6.1 Extending classes and class hierarchies with new behavior

To extend a class with new functionalities (e.g. new methods), subclassing is a natural choice. However, there are some limitations in its applicability. In particular, the new subclass is strongly coupled with its specific parent class, making it harder to apply such an extension to a group of classes. Decorators [GHJV95] and mixins [BC90] are possible solutions. However, mixins (or abstract subclasses) are not currently supported in Java (a class cannot extend a type parameter), and neither mixins nor decorators are able to replicate an important mechanism of a hierarchy of subtype related classes: the possibility in any class to override methods to a more specific or

efficient behavior.

To make this point clearer, consider the problem of extending shape classes (figure 3 in page 12) with a new test for convexity. For the sake of the argument, consider also that it was not possible to modify those classes. Clearly, the implementation of this new behavior depends on the specific shape: circles, rectangles and squares are always convex, but polygons might not be. So not only a mechanism is required that allows overriding methods to implement the correct algorithm, but also dynamic binding should be applicable (as in the original shape classes). Mixins or decorators are not a solution to this problem.

A single external dispatch provides a complete solution to the problem, with minimum extra code. Listing 12 shows a possible implementation.

A new `ExtendedShape` is created, as a subtype of `Shape` (hence, usable as a `Shape`). In this class, the overridden methods of `Shape` are implemented through delegation to the dispatch object. Also, we could have opted to give a default implementation to `isConvex` method in class `ExtendedShape` (for instance, returning true). In that case, we only need to redefine `ExtendedShape` descendants in which such property did not hold (in the example: `ExtendedPolygon` and `ExtendedExtendedShape`).

6.2 Creation design patterns

Creational patterns [GHJV95] allows the delegation of some responsibility on the type of the created object (or groups of objects), away from the creation request. For instance, abstract factories and factory method design patterns are characterized for abstracting away from the client, the concrete classes to be instantiated.

The external dispatch mechanism can provide the same behavior, requiring only a different dispatch argument type to enable the selection of external dispatch classes (where, under a common superclass, we can create the desired objects).

7 Final remarks

This article presents a new object-oriented external dispatch mechanism. Although carefully thought of, our proposal is still a work in progress. Syntactic and semantic changes may arise as a result of deeper formal analysis, compiler building, and testing.

References

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM. URL: `http://doi.acm.org/10.1145/97945.97982`, `doi:10.1145/97945.97982`.

[BC97] John Boyland and Giuseppe 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. `doi:http://doi.acm.org/10.1145/263698.263721`.

[BCC+95] Kim Bruce, Luca Cardelli, Giuseppe Castagna, Gary T. Leavens, and Benjamin Pierce. On Binary Methods. *Theor. Pract. Object Syst.*, 1(3):221–242, December 1995. URL: http://dl.acm.org/citation.cfm?id=230849.230854.

[BCV07] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Featherweight java with multi-methods. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 83–92, New York, NY, USA, 2007. ACM. URL: http://doi.acm.org/10.1145/1294325.1294337, doi:10.1145/1294325.1294337.

[BKK+86] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: merging Lisp and object-oriented programming. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 17–29. ACM Press, 1986. doi:http://doi.acm.org/10.1145/28697.28700.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, Cambridge, MA, USA, 2002.

[Cha92] Craig Chambers. Object-Oriented Multi-Methods in Cecil. In O. Lehrmann Madsen, editor, *Proceedings ECOOP'92, LNCS 615*, pages 33–56, Utrecht, The Netherlands, Jun 1992. Springer-Verlag.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd 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. doi:http://doi.acm.org/10.1145/353171.353181.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GJS+14] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in COMMON LISP*. Addison-Wesley, 1989. A Programmer's Guide to CLOS.

[KRLS13] Jieung Kim, Sukyoung Ryu, Victor Luchangco, and Guy L. Steele. *Fine-Grained Function Visibility for Multiple Dispatch with Multiple Inheritance*, pages 156–171. Springer International Publishing, Cham, 2013. URL: http://dx.doi.org/10.1007/978-3-319-03542-0_11, doi:10.1007/978-3-319-03542-0_11.

[LM98] Gary T. Leavens and Todd 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. doi:http://doi.acm.org/10.1145/286936.286977.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[OMM+04] Martin Odersky, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and et al. An overview of the scala programming language. Technical report, 2004.

[Par13] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[Ste90] Guy L. Steele, Jr. *Common LISP: The Language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990.

About the author

Miguel Oliveira e Silva is an assistant professor and researcher at University of Aveiro in Portugal. Contact him at `mos@ua.pt`, or visit `sweet.ua.pt/mos`.

Acknowledgments The writing of this article has benefited immensely from the critics of the anonymous reviewers, João Rodrigues, and Tomás Oliveira e Silva. Supported by Portuguese Foundation for Science and Technology (UID/CEC/00127/2013, Incentivo/EEI/UI0127/2014)