STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LXI**, Number 1, 2016

C# EXTENSION METHODS VERSUS JAVA DEFAULT METHODS IN THE CONTEXT OF MIXDECORATOR PATTERN

VIRGINIA NICULESCU

Abstract. Decorator design pattern is a very well-known pattern that allows additional functionality to be attached to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. *MixDecorator* is an enhanced variant of the *Decorator*, which does not just eliminate some constraints of the original one, but also allows it to be used as a base for a general extension mechanism. This pattern introduces significant flexibility by allowing direct access to all added responsibilities.
MixDecorator implementation imposes some constraints and we analyze and compare the implementation solutions in two of the most important mainstream object-oriented languages: Java and C#. This also leads to a comparison analysis between C# extension methods and Java default methods, or virtual extension methods as they are also called, in a more general context of trait-based programming.

1. Introduction

The authors of GoF Design Patterns book [2] consider that 'delegation' is an extreme form of object composition that can always be used to replace inheritance. *Decorator* pattern is one of the patterns that express well exactly these issues.

There are many situations when we want to add a combination of additional capabilities onto an object. However, the additional capabilities we want could be highly variable, and could extend the interface of the base object. We also may want all additional responsibilities to be directly available.

The classical *Decorator* pattern offers a solution to extend the functionality of an object in order to modify its behavior. It is usually agreed that decorators and the original class object share a common set of operations.

Still, when we want to add new responsibilities, and not just to change the behavior of the existing ones, the classical *Decorator* pattern allows us to define such a decoration, but the new responsibilities are accessible only if they are defined in the last added decoration.

Received by the editors: January 2016.

2010 *Mathematics Subject Classification.* 68P05.

1998 *CR Categories and Descriptors.* D.1.5 Object-oriented Programming D.3.3Language Constructs and FeaturesPatterns E.1DataData Structure .

Key words and phrases. object-orientation, patterns, decorator, responsibilities, languages, Java, C#.

MixDecorator pattern [3] does not just eliminate some constraints of the classical pattern (e.g. limitation to one interface), but also allows it to be used as a base for a general extension mechanism. Using it, we may combine different responsibilities, have direct access to all, and operate with them in any order.

We present in this paper a comparison analysis between the implementations of this pattern – as a case study – in two of the most important mainstream object-oriented languages: Java and C#. *MixDecorator* imposes some constraints and C# extension methods and Java default methods, need to be used.

This analysis represents a base for a more general analysis between C# extension methods and Java virtual extension methods, or default interface methods as they are also called, in the more general context of trait-based programming.

The paper is structured as follows: the second section briefly describes extension methods in C# and Java, and the next section succinctly presents the *Decorator* pattern and emphasizes the constraints imposed by the classical version of it. Section 4 describes the *MixDecorator* pattern. In the following two sections the implementations in Java and C# are discussed; a comparison analysis of these two is done in section 7. Conclusions and future work are presented in section 8.

2. Extension Methods in C# and Java

Java 8 introduces *default methods* in interfaces; they are also called virtual extension methods. The primary intent of this feature was to allow interfaces to be extended over time while preserving backward compatibility. Implicitly, interfaces in Java provide multiple type-inheritance, in contrast to class-inheritance. Still, Java 8 interfaces introduce a form of multiple implementation inheritance, too. A default method is a virtual method that specifies a concrete implementation within an interface: if any class implementing the interface will override the method, the more specific implementation will be executed. But if the default method is not overridden, then the default implementation in the interface will be executed[6]. It is also considered that Java 8 interfaces can be exploited to introduce a trait-oriented programming style [1].

In C# we have a mechanism called "extension methods" that allows adding new methods to a class after the complete definition of the class [5]. They allow the extension of an existing type with new functionality, without having to sub-class or recompile the old type. The mechanism allows only *static binding*, and so the methods that could be added to a class cannot be declared virtual. In fact, an extension method is a static method defined in a non-generic static class, and can be invoked using an instance method syntax.

3. Decorator pattern and its constraints

The Decorator pattern is a structural pattern used to extend or alter the functionality of objects at run-time by wrapping them in an object of a decorator class. This pattern is designed such that multiple decorators can be stacked on top of each other, each one adding new functionality to the overridden method(s) [2].

This means that objects based upon the same underlying class can be decorated in different ways. In addition, as both the class of the object being modified and the

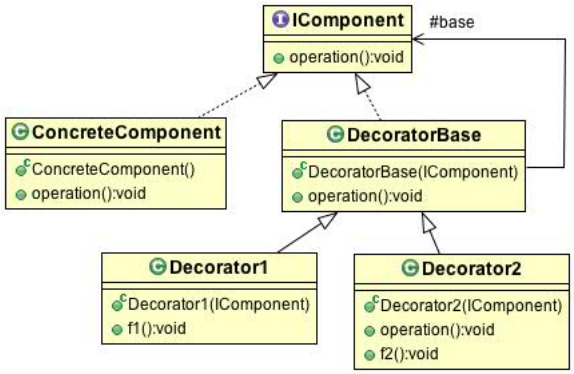

FIGURE 1. The class diagram of the standard Decorator pattern.

class of the decorator share a base class, multiple decorators can be applied to the same object to incrementally modify behavior.

3.1. **Limitations of the classical Decorator pattern.** As a possible usage scenario we may consider that we have n responsibilities intended to be defined as decorations for a base class `IComponent`. These responsibilities are defined as methods – `f1, f2, ..., fn`. As the pattern specifies, n decorator classes will be defined (`Decorator1, Decorator2` ... `Decorator`n), each defining the corresponding method, and they are all derived from a decoration class `DecoratorBase`, which is in turn derived from `IComponent`. Theoretically, we may obtain any combination of decorations but we only have the base class interface available.

So, if there are some responsibilities that are really new responsibilities (that change the object interface) and they are not used just to alter the behavior of the operations defined in the base class, they will be accessible only if the last decoration is the one that defines them. We will refer to this kind of decorations as *interface responsibilities*. More concretely, if responsibility `f1` is a new interface responsibility and it is defined in the class `Decorator1`, then the corresponding message could be sent only to an object that has the `Decorator1` decoration, but also only if this is the last added decoration.

4. MIXDECORATOR

By using *MixDecorator* we are able to attach *a set* of additional responsibilities to an object dynamically, and to allow direct access to all added responsibilities. It provides an alternative to subclassing for extending objects functionality and their types also (by extending the set of messages that could be sent to them) [3].

The solution of the *MixDecorator* is inspired by the *Decorator* but there are several important differences. As for simple decorators we enclose the subject in another

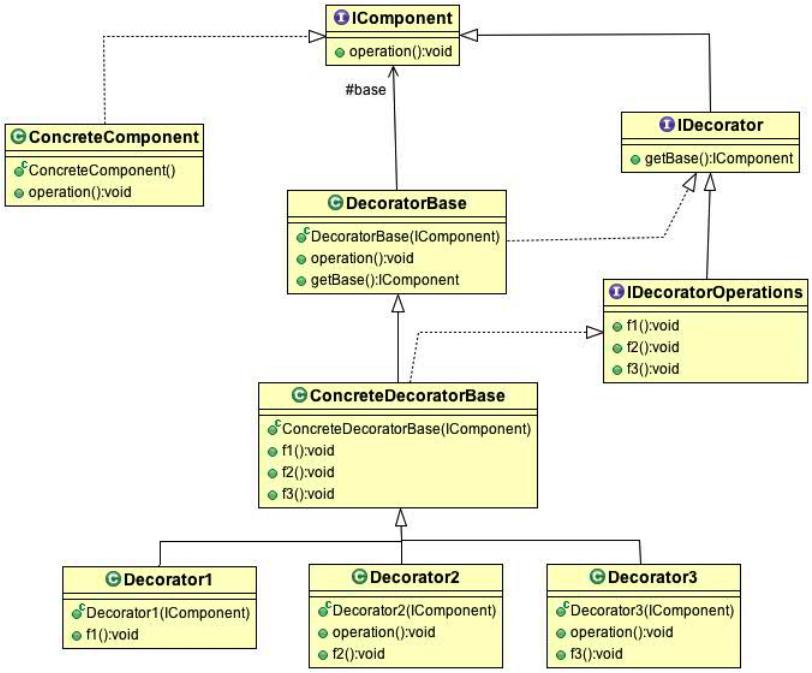

FIGURE 2. The class diagram for the MixDecorator pattern.

object, the decorator object, but the decorator could have an interface that extends the base interface.

The general solution is presented in Figure 2. This makes a clear separation between `IComponent` and `DecoratorBase` by introducing a general `IDecorator` interface that extends `IComponent` and adds only `getBase()` method (this method is considered mandatory). The concrete class `DecoratorBase` has almost the same definition as the corresponding class from the classical *Decorator* (the difference is the additional method `getBase()`).

For a particular application, after the new responsibilities are inventoried, then, particular `IDecoratorOperations` and `ConcreteDecoratorBase` are defined. `IDecoratorOperations` defines the methods that correspond to all new responsibilities. `ConcreteDecoratorBase` is derived from `DecoratorBase` but also implements `IDecoratorOperations`. The interface is introduced in order to emphasize the new added resposibilities.

The following code snippet emphasizes the forces fulfilment; the execution throws no exception, and it can be noticed that, for example, `f3()` could be called even if `Decorator3` is not the last added decoration.

```
1    IComponent c = new ConcreteComponent();
2    IDecoratorOperations d = new Decorator1(new Decorator2(new Decorator3(c)));
```

```
3     d.operation();
4     d.f3();  d.f1();  d.f2();
```

This could be achieved because in `ConcreteDecoratorBase` class we define a *"recursive search"* of the methods, as it is emphasized by the following Java code:

```
1  public class ConcreteDecoratorBase extends DecoratorBase implements
       IDecoratorOperations {
2       public ConcreteDecoratorBase(IComponent base)
3       { super(base); }
4       public void f1() throws UnsupportedFunctionalityException{
5           IComponent base = this.getBase();
6          if ( base instanceof IDecoratorOperations)
7                ((IDecoratorOperations)base).f1();
8          else //if base is not a decorator but a concrete component
9                throw new UnsupportedFunctionalityException("f1");
10      }
11      . . .
12 }
```

(Direct casting inside a try-catch block could be used instead of `instanceof` operator).

The base case of solving a call is when the invoked responsibility is defined in the last added decoration: in this case the object directly calls the method. If the invoked responsibility is not defined in the last added decoration, then its definition from `ConcreteDecoratorBase` is used. The recursion is stopped when the obtained `base` is just a simple component, and not a decorator that implements `IDecoratorOperations`.

The corresponding implementation in other object-oriented language, is similar.

4.1. **Extensions with other responsibilities.** If other possible responsibilities are discovered as being appropriate to be used, these could be added in general using the following steps:

(1) Define a new interface – `IDecoratorOperations_Extended` that extends the interface `IDecoratorOperations` interface, and defines the desired new responsibilities.
(2) Define a class – `ConcreteDecoratorBase_Extended` that extends `ConcreteDecoratorBase` and implements `IDecoratorOperations_Extended`.
(3) (optional) Provide an adaptation that assures that all the responsibilities either added in the first design iteration or in the next, could be combined in any order.

Figure 3 illustrates the new added classes.

The specified adaptation as the third step could be done using, for example, *Adapter* pattern. The previous decoration classes are adapted to the new extended interface. For example the class `Decorator2_Adapted` is derived from `Decorator2`, and implements `IDecoratorOperations_Extended`; no method overriding is necessary. But, this extension also requires a basic implementation of the methods defined in the interface `IDecoratorOperations_Extended`. (Without the adaption we can wrap the initial set of decorations with the new ones, but viceversa is not possible.)

The implementation of the class `ConcreteDecoratorBase_Extended` is similar to

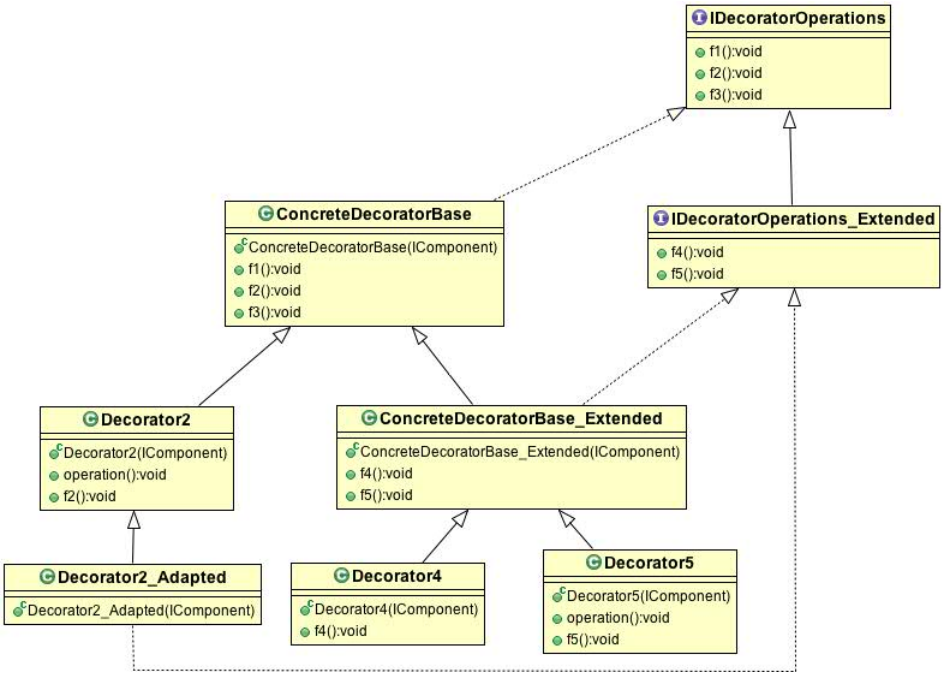

FIGURE 3. The classes that need to be defined when new decorations are intended to be added.

that of `ConcreteDecoratorBase`. Next, a usage example based on the presented structure is given:

```
1          IComponent c = new ConcreteComponent();
2          IDecoratorOperations d31 = new Decorator3(new Decorator1(c));
3          IDecoratorOperations_Extended d431 = new Decorator4(d31);
4          IDecoratorOperations_Extended d2431 = new Decorator2_Adapted(d431);
5          d431.f3(); d431.f4(); d431.f1(); d2431.f2(); d2431.f4();
```

The code produces the correct execution of all the methods.

Generally, in order to allow new decoration extensions, there is an implementation requirement defined by the possibility of adding new methods to an interface (to add a set of methods to `IDecoratorOperations` interface), and also to provide a basic implementation for them.

Classically, this is done based on multiple inheritance. So, a language as C++ or any other that allows multiple inheritance leads to a simple implementation, where `IDecoratorOperations_Extended` is defined as an abstract class. Other mechanisms – specific to the target language – could be investigated. An example is provided by the Java default methods.

5. Java Implementation

In Java, a simplified implementation of *MixDecorator* is possible by using interface default methods. The class `ConcreteDecoratorBase` could be eliminated, and the methods declared in the `IDecoratorOperations` interface could be defined as default methods, with the corresponding implementations taken from `ConcreteDecoratorBase`. Figure 4 emphasizes the specific class diagram, and the following code snippet some implementation details.

```
public interface IDecoratorOperations extends IDecorator{
       default public void f1() throws UnsupportedFunctionalityException{
              IComponent base = getBase();
              if(base instanceof IDecoratorOperations){
                     ((IDecoratorOperations)base).f1();
              }
              else throw new UnsupportedFunctionalityException("f1");
       }
  . . .
 }//~ end of the interface IDecoratorOperations
```

(This solution is based on operator `instanceof`, but type-casting could be used instead.)

Defining new decorations - and so extending the functionality - could also be simplified in Java implementation.

The class `ConcreteDecoratorBase_Extended` should not be defined anymore, since again default methods could be used for interface `IDecoratorOperations_Extended`. The implementation of the method `f4()` in `IDecoratorOperations_Extended` could be defined in Java as:

```
public interface IDecoratorOperations_Extended extends IDecoratorOperations{
  default void f4() throws UnsupportedFunctionalityException{
    try{ ((IDecoratorOperations_Extended)getBase()).f4(); }
    catch(ClassCastException e){
       throw new UnsupportedFunctionalityException("f4"); }
  }
 . . .
}//~ end of the interface IDecoratorOperations\_Extended
```

(This solution is based on type-casting, but operator `instanceof` could be used instead.)

If we need to define a new decoration that overrides a responsibility (method) defined by a previous decoration, this is possible by defining the new decoration class as a subclass of the initial decoration class that defines the new responsibility. It was the case of `Decorator4_Second` that extends `Decorator4` and overrides `f4()`. Since the Java solution is based on polymorphic calls, if the used decoration is `Decorator4_Second`, then the method `f4()` defined in this class is used.

Also, in Java, we may simplify the implementation for extensions by replacing the implementation of the interface `IDecoratorOperations` with a new one that defines the new methods too (methods `f4(), f5()`). In this way, the new Java mechanism is used to the maximum efficiency. The initial decoration classes does not have to

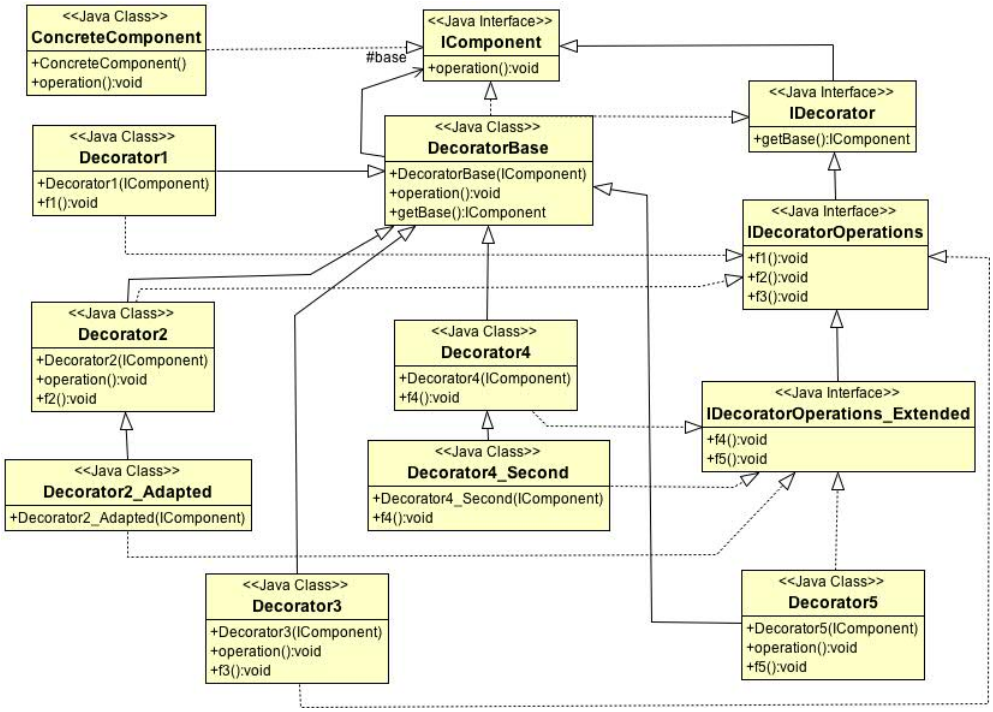

FIGURE 4. The class diagram for Java implementation of the *MixDecorator* pattern.

be recompiled, and no adaptation is needed. Still, we need to have access to the `IDecoratorOperations` interface in order to replace its implementation.

6. C# IMPLEMENTATION

In C# the simplification could be done by adding the new decoration methods directly to the interface `IDecorator`, and by excluding both `ConcreteDecoratorBase` and `IDecoratorOperations`. Figure 5 presents the corresponding class diagram.

New static classes that define the extensions methods for `IDecorator` should be introduced (e.g. `Decorator_Extension12` with the methods `f1(), f2()`). The main difference of this solution is that being based on static methods, the base case should be treated inside the extension method, too. The extension methods define a searching recursive mechanism for each new method.

Also, we still may use the type `IDecorator` for the wrapped objects, that, in this case, will accept messages defined as new methods in the decorators.

```
1  public static class Decorator_Extensions12
2     { public static void f1(this IDecorator cdb)
3         {try
```

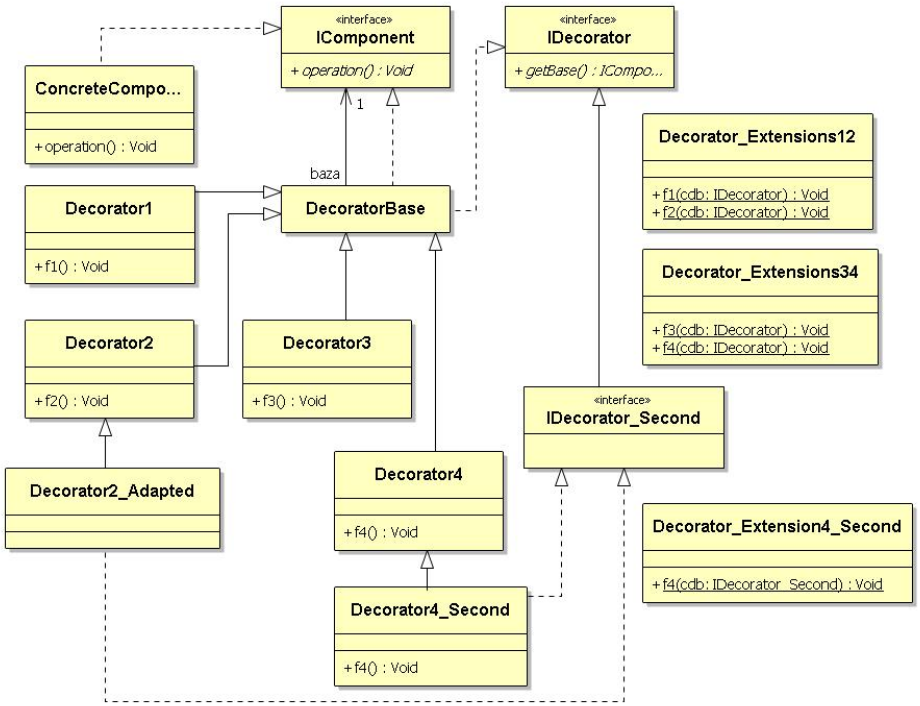

FIGURE 5. The class diagram for C# implementation of the *MixDecorator* pattern.

```
4           { ((Decorator1)cdb).f1(); }
5           catch (InvalidCastException e)
6           { try { ((IDecorator)cdb.getBase()).f1(); }
7               catch (InvalidCastException ee)
8               { throw new UnsupportedFunctionalityException("f1"); }
9           }
10          . . .
11      }//~ end of class Decorator_Extension12
```

(This solution is based on type-casting, but operator *as* could be used instead; also the exception type `UnsupportedFunctionalityException` could be replaced by the C# exception type `System.NotSupportedException`.)

The extension methods define each a recursion that it's stopped either if the current decoration is that one which defines the invoked method or by throwing an exception if no decoration that defines such a method was found.

If other decorators are added (`Decorator3, Decorator4`), other extension class could be defined in a similar way. With this C# solution, no adaptation of the first defined decoration classes is needed, because in fact there is no difference between defining new responsibilities in the first design iteration, or in the next.

But this simple solution doesn't works well in all the cases:
If more decorations are added, which define, for example, a responsibility with the same name as a previously defined responsibility (for example in Figure 5 there is also `Decorator4_Second` that defines a method `f4()`), there are two possibilities to solve this problem:

- **-:** To change the implementation of the method `f4()` which was defined in the class `Decorator_Extension`, such that it will verify in chain all the possibilities (starting from the more specialised class).
- **-:** To define a new interface `IDecorator_Second` that extend `IDecorator` interface, make `Decorator4_Second` implements this interface, and define a new extension class `Decorator_Extension4_Second` where the new definition for `f4()` could be added. This variants implies also an adaptation for all previously defined decorator – as it was described for the general case and for Java implementations (Figure 5 emphasizes this solution).

7. Comparison of Java and C# solutions

Both implementations, allow simplification of the general solution.

The C# extensions methods are static methods that are called as if they were instance methods. This static character does not affect the solution in the simple case (when there are no overridings of the new added responsibilities) because, these static methods provide just searching mechanisms for the instance methods with the same name. But, just in these simple cases, the searching mechanism does not have to be changed dynamically. Since all the extensions are done on the same interface `IDecorator` we do not need to adapt the decoration classes to new interfaces (in case of a multistage development).

Java implementations is similar to an implementation based on multiple inheritance that can be used in any object oriented language that accept multiple inheritance (ex. C++). Java 8 default methods implies a form of multiple inheritance (behavior multiple inheritance). The Java solution is efficient and the possible simplifications over the general case are important. The advantage is given by the dynamic bindings of virtual extension methods, that simplifies the searching mechanism; the base case – when the front decoration is the one that defines the called responsibility – is solved automatically. The disadvantage is represented by the need of adaptation of the previous decorations to the new added set.

Apparently the implementation of *MixDecorator* pattern is simpler in C#, but if we thoroughly analyze the solutions we can notice that for the general case, when the new responsibilities could be overriden, the C# mechanisms based on static binding (imposed by the extension methods) in searching mechanism implies an explicit specification of the type where the corresponding method (specified by the message name) is defined. This imply to rewrite the extension method that correspond to a message name, each time a new decoration that overrides the corresponding method is defined, or to define a new decorator interface, together with adaptations of the previously defined decorators.

The searching algorithm for the concrete implementation of a responsibility is the same for all the responsibilities and so, we may try to use *Template Method* pattern [2]. This means to define a method `execute` that receive the name of the responsibility and a variable list of arguments, and tries to call the concrete methods. The invocation should be done based on reflection since the name of the method is given as a string. The recursive search of the method could be included inside this method `execute`.

If we want to provide a general extension mechanism similar to that provided by the Scala `traits`, then an implementation of *MixDecorator* based on a general dispatcher could be used.

8. Conclusions

MixDecorator – is similar to *Decorator* pattern in the sense that allows functionality extension, but it treats also the situations when we want to add new responsibilities, more concretely, when we want to enlarge the set of messages that could be sent to an object (so we may consider that we dynamically modify the type of an object). The structure of the pattern as emphasized in Figure 2 could be easily implemented in any object-oriented language. But in Java 8 and C# its implementation could be simplified.

Essentially, the implementation requirement is defined by the possibility of adding new methods to a class. Classically this is done based on inheritance. Other mechanisms – specific to the target language – could also be used.

Two implementations – in Java and C# – were analyzed, and this analysis emphasizes the differences and also the advantages and disadvantages of Java extended interfaces and C# extension methods.

The C# extension methods could be considered as a weaker mechanism over Java virtual extension methods (default methods) since they do not allow polymorphism. Still, in the context of implementing *MixDecorator* pattern they lead to a very simple solution, for the case when no method overrinding is used. Java implementation is a more object-oriented solution since some actions are done implicitly based on polymorphic call of the invoked methods. C# extension methods provide a very flexible way to add a method to a class, but as the C# developers say "extension methods certainly are not pure object-oriented". Still, extension methods are features of some object-oriented programming languages, most of them being dynamic languages.

References

[1] V.Bono, E. Mensa, M. Naddeo. *Trait-oriented Programming in Java 8.* PPPJ'14: International Conference on Principles and Practices of Programming on the Java Platforms, Sep 2014, Cracow, Poland.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1994.

[3] V. Niculescu *MixDecorator: An Enhanced Version of Decorator Pattern* In Proceedings 20th European Conference on Pattern Languages of Programs (EuroPLoP'2015) Kloster Irsee, Germany 8-12 July 2015, Art No. 36 (doi:10.1145/2855321.2855358).

[4] A. Shalloway, J. R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design.* Addison Wesley, 2004

[5] *Extension Methods (C# Programming Guide)*
https://msdn.microsoft.com/en-us//library/bb383977.aspx
[6] *Java SE 8: Implementing Default Methods in Interfaces*
https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html/

Department of Computer Science, Babeş-Bolyai University, Kogalniceanu 1, 400084, Cluj-Napoca, Romania
E-mail address: vniculescu@cs.ubbcluj.ro