MIXDECORATOR: AN ENHANCED VERSION OF DECORATOR PATTERN

VIRGINIA NICULESCU,
BABEŞ-BOLYAI UNIVERSITY, CLUJ-NAPOCA

Abstract. Decorator design pattern is a very well-known pattern that allows additional functionality to be dynamically attached to an object. Decorators provide a flexible alternative to subclassing for extending functionality. In this paper we analyse and design a pattern – *MixDecorator* – that could be considered an enhanced version of the Decorator pattern, which does not just eliminate some constraints or limitations of the original one, but also allows it to be used as a base of a general extension mechanism. This pattern introduces significant flexibility by allowing direct access to all added responsibilities. Using it, we may combine different responsibilities and operate with them directly and in any order. A complex example of using it for a collection framework design is presented.

Keywords: OOP, design patterns, decorator, responsibility, extensibility, features, framework

1. Introduction

The authors of *Gang of Four* Design Patterns book [GHJV94] argue in favor of *object composition* over *class inheritance*. To the authors, '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.

We present in this paper an enhanced version named – *MixDecorator* – of the classical *Decorator* pattern defined in [GHJV94]. The classical *Decorator* pattern offers a solution to extend the functionality of an object in order to modify its behavior. Still, when we want to add new responsibilities, and not just to change the behavior of an existing one, the classical *Decorator* pattern allows us to define such a decoration, but this is accessible only if it is the last added one. The presented enhanced version 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. This version introduces significant flexibility and abstraction. Using it, we may combine different responsibilities, have direct access to all, and operate with them in any order.

The paper is structured as follows: next section succinctly describes the classical version of the *Decorator* pattern and emphasizes the constraints imposed by it. Section 3 describes the new proposed pattern. Next, in section 4 a complex example, which is designed using the new proposed pattern, is presented. The example consists of a collections framework for which we consider features based definitions of collections.
Conclusions and future work are presented in section 5.

2. Decorator pattern

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 provides a flexible alternative to using inheritance for

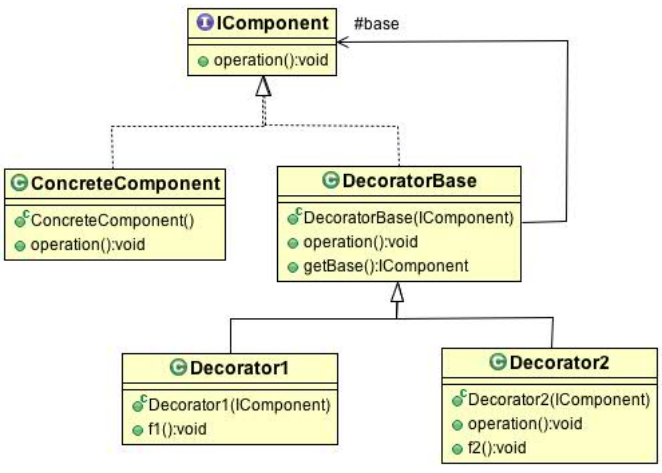

Figure 1. The class diagram of the standard Decorator pattern.

modifying behavior. *Decorator* 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). Figure 1 shoes the corresponding class diagram. It is usually agreed that decorators and the original object's class share a common interface. This means that objects based upon the same underlying class can be decorated in different manners. In addition, as both the class of the object being modified and the class of the decorator share a base class, multiple decorators can be applied to the same object to incrementally modify behavior[GHJV94, ST04].

2.1. **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 `DecoratedBase`, 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 changes 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; the following Java code snippet emphasizes this:

```
1 IComponent o = new Decorator1(new Decorator2(new ConcreteComponent()));
2         ((Decorator1).o).f1();
3 IComponent oo = new Decorator2(new Decorator1(new ConcreteComponent()));
4         // ((Decorator1).oo).f1(); ERROR
5        ((Decorator1)oo.getBase()).f1();   //an improper  solution
```

The emphasized solution has obviously several drawbacks:

- we have to apply an additional operation that allows decoration removal – `getBase()`; in case we don't have it, there is no solution;
- the functionality added by `Decorator2` is lost (the behavior modification of the base operations brought by `Decorator2`) ;
- if there are several decorations that should be removed then several additional operations are necessary;
- if we don't know the exact position (order) of the searched decoration, the code becomes very complex (a kind of reflection should be used);

In fact, removing the last decorations in order to reveal functionality is not a real solution since it breaks the way in which decorated objects are supposed to be used. Also, it is an ad-hoc workaround that is based on knowing the order in which decorations were added.

For example, we may consider the classes in Java IO streams library, where *Decorator* pattern is used. `FilterInputStream` corresponds there to `DecoratorBase` and it's derived from `InputStream` that corresponds to `IComponent`. There are several decoration classes derived from `FilterInputStream` such as `PushBackInputStream` that defines a method `unread()`, which is not defined in the `FilterInputStream` interface; `BufferedInputStream` that just alters the behavior of the standard `InputStream` interface; or `CheckedInputStream` that maintains a checksum of the data being read and allows using it based on the method `getChecksum`.

We may combine them, and decorate a concrete stream - e.g. `FileInputStream` - first with `CheckedInputStream` and then with `BufferedInputStream` or/and with `PushBackInputStream`; then the `getChecksum` method is not directly available.

```
1 PushbackInputStream pi = new PushbackInputStream(
2         new BufferedInputStream(
3                 new CheckedInputStream(  new FileInputStream("input"), new CRC32())));
4 //      pi.getChecksum(); ERROR
```

Since, the class `FilterInputStream` does not provide an operation as `getBase()` not even the simplistic solution presented before is not possible. (The class `FilterInputStream` has a field `in` but it is protected and so inaccessible; a solution could be to derive all the classes derived from `FilterInputStream` and define for them a method `getBase()` method that returns `in`.)

3. MixDecorator Pattern

3.1. **General Definition.** We specify the characteristics of the *MixDecorator* pattern by emphasizing in detail what is specific to it.

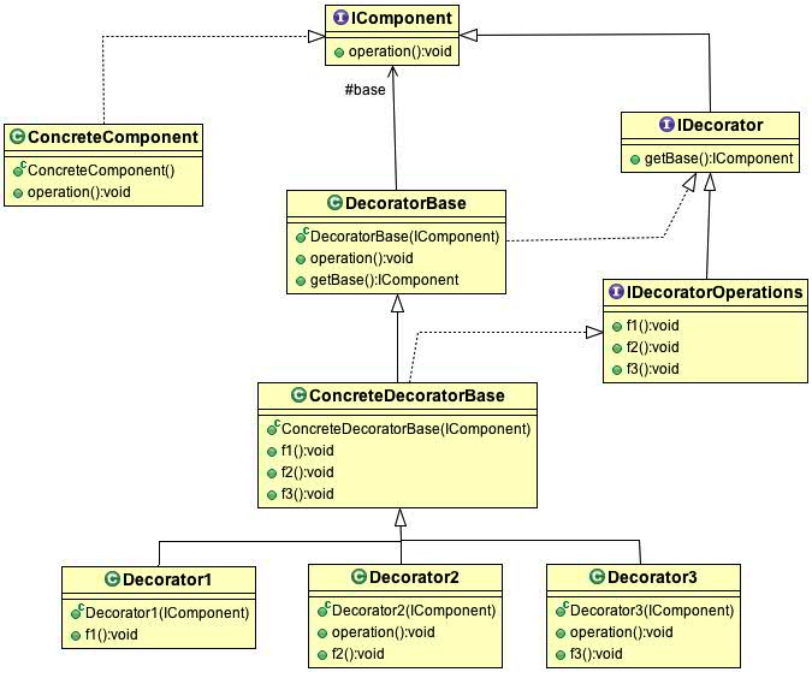

FIGURE 2. The class diagram for the MixDecorator pattern.

3.1.1. *Synopsis.* Attach *a set* of additional responsibilities to an object dynamically. Allow access to all added responsibilities. Provide a flexible alternative to subclassing for extending objects functionality and their types, too (extending the set of messages that could be sent to them).

3.1.2. *Context.* You want to add a combination of additional capabilities onto an object. However, the additional capabilities you want are highly variable, and could extend the interface of the base object. You want all additional responsibilities to be directly available.

3.1.3. *Problem.* The classical *Decorator* pattern offers a solution to extend (decorate) the functionality of a certain object in order to modify its behavior. It is usually agreed that decorators and the original class object share a common interface. The problem appears when we want to add a new interface responsibility, and not just to change the behavior of an existing one. Such a decoration could be defined, but it is accessible only if it is the last added one.

3.1.4. *Forces.*

- Adding responsibilities should be transparent to clients.
- All responsibilities which are added at some point are directly accessible to the client.
- Simple and direct call for all decoration methods, not dependent of the exact position (order) of the decoration where the desired responsibility is defined.
- It should be easy to change, e.g. withdraw or adding, responsibilities.
- Good efficiency, and easily extendable.

3.1.5. *Solution.* The structure of the *MixDecorator* is inspired by the *Decorator* but there are several important differences that allow the achievement of the *'forces'*. As for simple decorators we enclose the subject in another object, the decorator object, but the decorator could have an interface that extends the general component interface. The decorator forwards requests to the subject while performing additional actions before and after forwarding.

The solution structure 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 `DecorateBase` 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. As it can be seen from Figure 2, `ConcreteDecoratorBase` is derived from `DecoratorBase` but also implements `IDecoratorOperations`.

In order to better explain the pattern we will give some implementation details in Java 8.

The concrete class `ConcreteDecoratorBase` gives a definition for each new added responsibility. It implements `IDecoratorOperations` and extends `DecoratorBase`. The corresponding code hides a recursion that is used for searching the method (for example `f1()`) that defines the new responsibility.

The corresponding Java code is:

```
public class ConcreteDecoratorBase extends DecoratorBase implements IDecoratorOperations {
       public ConcreteDecoratorBase(IComponent base)
       {  super(base); }
       public void f1() throws UnsupportedFunctionalityException{
          try{
               ((IDecoratorOperations)getBase()).f1(); }
          catch (ClassCastException e){//if base is not a decorator but a concrete component
              throws  new UnsupportedFunctionalityException("f1");
          }
      }
      . . .
}
```

The following code snippet emphasizes the forces fulfillment; the execution throws no exception, and it can be noticed that, for example, `f3()` could be called even if `Decorator3` is the first added decoration.

```
     IComponent c           = new ConcreteComponent();
     IDecoratorOperations d = new Decorator1(new Decorator2(new Decorator3(c)));
     d.operation();
     d.f3();            d.f1();         d.f2();
```

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

(1) Define a new interface – `IDecoratorOperations_Extended` that extends `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 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 also requires a basic implementation of the methods defined in `IDecoratorOperations_Extended`.

Java 8 introduces default methods in interfaces. Implicitly, interfaces 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[Java8].

The implementation of the method `f4()` in `IDecoratorOperations_Extended` could be defined in Java as:

```
default void f4() throws UnsupportedFunctionalityException{
  try{ ((IDecoratorOperations_Extended)getBase()).f4(); }
  catch(ClassCastException e){
       throw new UnsupportedFunctionalityException("f4"); }
}
```

This definition assures that the first set of decorations could be combined with the new decorations in any order, inclusive to wrap around the new decorations (optional step (3)).

The implementation of the class `ConcreteDecoratorBase_Extended` is similar to that of `ConcreteDecoratorBase`.

Next, a usage example based on the presented structure is given:

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

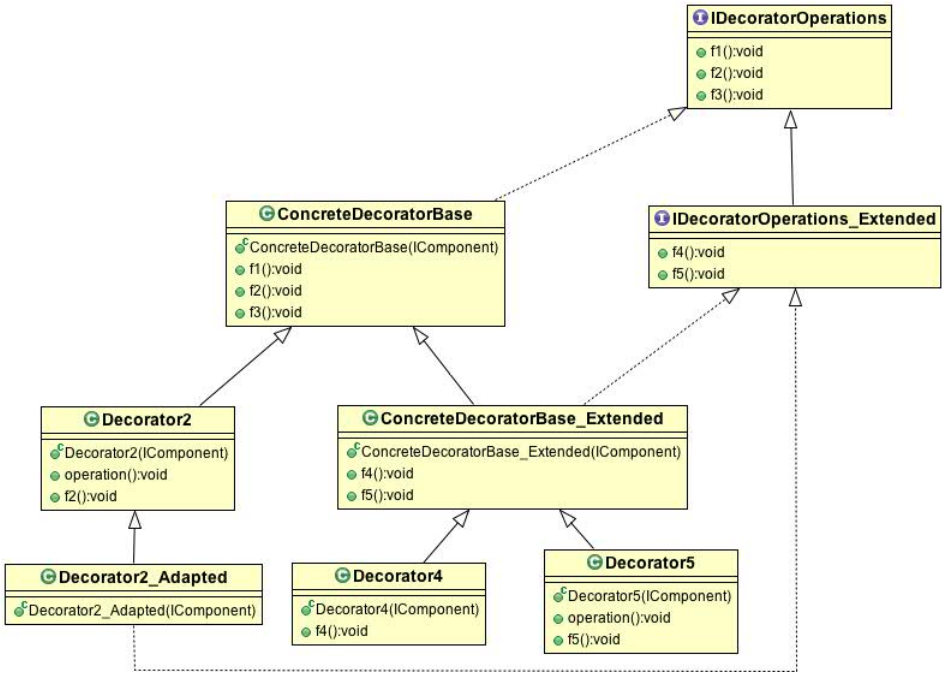

FIGURE 3. The classes that need to be defined when new decorations are intended to be added.

The code produces the correct execution of all the methods.

The set of the possible added responsibilities could be extended easily.

3.3. *Implementation.* The structure of the pattern as emphasized in Figure 2 could be easily implemented in any object-oriented language.

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 lead 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 extended interfaces (or virtual extension methods- as they are also called). They are based on defining default implementations inside interfaces (the implementation details for our case were given in the previous section). It is also considered that Java 8 interfaces can be exploited to introduce a trait-oriented programming style [BMN14].

Also, the implementation of the *MixDecorator* could be simplified by considering only `IDecoratorOperations` without `ConcreteDecoratorBase`. In Java 8 this would be an interface with default methods (with the same definitions as they are now in `ConcreteDecoratorBase`).

In C# the pattern could be implemented based on *extension methods*. Using "extension methods" we are able to add new methods to a class after the complete definition of the class [CS]. 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.

If we use them in C#, we may add a static class `Decorator_Extension` where the methods `f4(), f5()` are defined as extension methods. The class `Decorator_Extension` provides extension for `IDecoratorOperations`:

```
1     public static class Decorator_Extension{
2      public static void f4(this IDecoratorOperations db)  {
3           try
4            {  ((Decorator4)db).f4();  }
5           catch (InvalidCastException e)
6              {   try {  ((IDecoratorOperations )db.getBase()).f4();  }
7                  catch (InvalidCastException ee)
8                         { throw new UnSupportedFunctionalityException("f4");} }
9         }
10  ...    }
```

The code defines a recursion that it's stopped either if the current decoration defines the invoked method or by throwing an exception if no decoration that defines such a method was found.

In Java, the verification of the first case is implicitly done based on polymorphic call.

With this C# solution, no adaptation of the first decorator classes is needed.

Next, we show an example of combining decorations in C#; it produces correct calls:

```
1 IComponent c           = new ConcreteComponent();
2 IDecoratorOperations d = new Decorator 2(new Decorator5(new Decorator1(new Decorator4(c))));
3 d.f1(0; d.f2(); d.f4(); d.f5();
```

Applications. The applications that were designed based on the classical *Decorator*, and which define new responsibilities for the decorated objects, could be improved by using *MixDecorator*.

The definition of Java IO streams is such an example. If a method of type `getBase()` would be provided in `FilterInputStream` then *MixDecorator* could be used instead of simple *Decorator*, and so we could eliminate the constraints of the current implementation.

Many other similar situations could be found, where decorations imply also adding new operations.

3.4. *Consequences.*

- The linear combination of the decorations is hidden. The final object could be seen as an object with a set of additional responsibilities. This way we may consider that we extend the type of an object (by extending the set of messages that could be sent to it) without defining new classes.
- The component and its decorators are decoupled. The author of the component does not need to do anything special for it to be decorated.
- Added behavior could be used in any combination, without any additional operation: such as withdrawing decorations.
- Clients can choose what capabilities they want by sending messages to the object that has the appropriate decoration.
- It is not necessary to create different decorated objects in order to choose a combination of capabilities, even if the decorators‘ concrete interfaces are not the same as the subject interface (but they are derived from it).
- As for classical decorators, objects do not pay for capabilities they do not use. Thus we have efficiency and generality at the same time.

Negative Consequences.

- If the *Decorator* pattern requires virtually no anticipation, in the case of *MixDecorator* some anticipation is required in the sense of knowing the initial set of decorations that is intended to be provided. The definition of `IDecoratorOperations` implies an enumeration of all new interface responsibility that is going to be defined through decorations. Still, even if initially just a small set of decorations are defined, it could be extended with minimum adaptation (see section 3.2) by defining new decorations.
- The addition of new decorations that could be combined to the previous ones imposes an implementation constraint which results from the following requirement – to allow adding new methods to an interface (to add a set of methods to `IDecoratorOperations`), and to provide a basic implementation for these methods.This could be directly implemented in languages that accept multiple inheritance – as C++, or using a surrogate of it, as that provided by the Java extended interfaces, or other mechanisms specific to the implementation language – as C# extension methods.

Related patterns. If the component class is heavyweight, with lots of data or methods, it may make decorators too costly. It is considered that instead of changing the skin of the object, we can change the guts, via the *Strategy* pattern.

Strategies do not have to conform to the subject's interface but the same is for *MixDecorator*, too. In the *Strategy* pattern, the main component is aware of the existence of strategies, where that is not needed with decorators. Also an important advantage of *MixDecorator* is the fact that allows behaviors mixing, when using *Strategy* we can just choose one strategy/behavior or another.

4. Example – Framework for Collection Data Structures with Features

We proposed in [NL13] a design based on *Decorator* pattern for frameworks of collection data structures. The implementation and testing of the proposed design has been done in Java. The analysis showed that *Decorator* has several drawbacks if it is applied in the classical way. The further analysis leaded to the *MixDecorator* pattern.

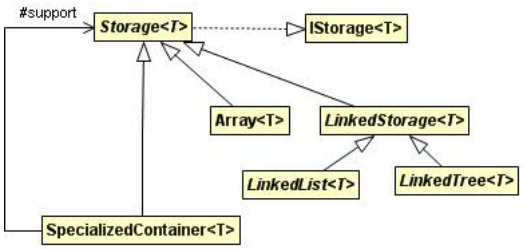

FIGURE 4. *Decorator* based design of the collections. `SpecializedContainer` is the Decorator class, and `IStorage` defines the general component. Concrete storages could be defined: `Array`, `LinkedStorage`, etc.

4.1. **General ideas.** We consider that there are two general and important aspects related to collections [Nic11]:

(1) *storage capability* – the elements that are grouped together have to be stored into the memory in an accessible way; usually the term *container* emphasizes more this aspect;
(2) *specific behavior* – the operations that are allowed for a specific type of container have different specifications; usually the term *collection* is chosen to emphasize this aspect.

Storage capability. The first aspect is directly connected to the data structures used for storing the elements. For storage, we may use a continuous block of memory or a set of discontinues blocks of memory (nodes) connected one to another using links (references). Each container has to be stored in the memory, in a way that allows elements to be added, removed, retrieved, and searched. The storage capability of a collection could be considered as a basic, compulsory, implicit feature, that characterizes any collection[Nic11].

Specialized behavior – specialized containers. The set of operations that could be applied to a container may be different, but also their specification may be different from one collection type to another. In order to emphasize these differences from behavior point of view, we may identify a set of features that could be applied to a container.

So, our approach is not based on abstract data types, but on specific behaviors defined with features. This was inspired by feature-oriented programming [BG97, CE00]. The design was leaded by the following main principle:

¡*Anytime a feature could be added to a collection and then could be removed.*¿

We considered *a feature* as being a distinctive property that characterizes the behavior of a collection – an operation or a set of operations with defined arguments, together with their semantic, expressed by a clear specification. It is something that fundamentally characterizes the collection behavior [NL13].

Starting from a concrete storage structure we may create different collection types, by adding different behaviors.

¡A *behaviour* is defined as being formed of a combination of basic features.¿

For example, a *set* is characterized only by the fact there are no duplicate elements in the container. The feature `Unique` defines the operation `add` with the same argument list as in the basic storage type, but changes the postcondition of the operation, by assuring the fact that the argument is added only if its value is not yet present in the container. How these elements are stored, is not a fact that characterizes the set.

`Sequence` assures the fact that the elements are in a particular linear order; the implementation translates this abstract specification into a concrete behavior by offering a bidirectional read and write iterator.

`Ranked` is a feature that specifies an added behavior that allows the access to the elements based on their rank. A rank of an element in a collection is equal to the rank of it in the traversal executed by the implicit iterator. So, this could be added not only to sequences.

Stacks and queues specify particular behaviors, and because of that, they could be seen as features. They are specializations of sequences.

Many other features could be defined, and this represents the main modality of extending the framework.

4.2. **Features classification.** The features could be classified depending on how they change the behavior of the container:

- features that preserve the default container operations, but changes their specifications; ex. `Unique` (symmetric features);
- features that add new operations; ex. `Ranked, Sequence`;
- features that restrain the set of operations; ex. `UnmodifiableStorage`;
- features that restrain the implicit set of operations, but add some other new operations; ex. `Stack, Queue` – they eliminate `remove(elem)`, and introduce `extract()`;

We may identify some restrictions; for example there are features, which could not be added after we have already have some elements into the support container. All these features are specializations of `EmptyStorage` feature. Generally, between features we may establish specialization/generalization relationships.

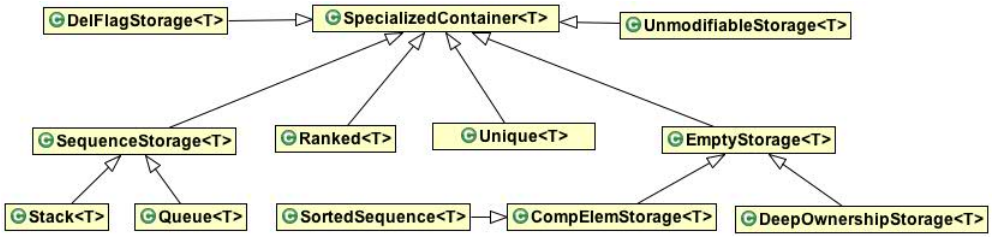

FIGURE 5. Different features that could be defined for a collection. The features are defined as decoration classes. `SpecializedContainer` is the base decorator class, which is derived from `Storage` but also wraps inside a `Storage` field.

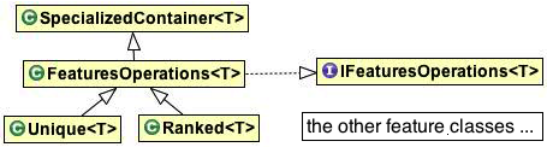

FIGURE 6. The design adaptation to *MixDecorator*.

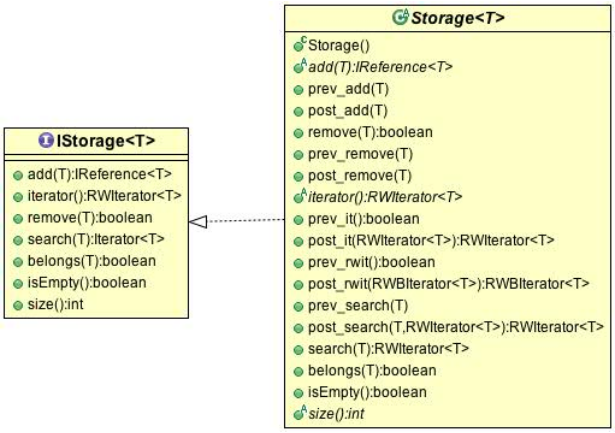

FIGURE 7. The `IStorage` interface and the methods of the class `Storage` .

The first design of the framework was based on *Decorator* pattern and this forced us to classify the features into levels and impose an order in the possible addition of the features based on these levels. `Ranked`, for example, adds operations like `getElem(index)`, `setElem(val,index)` , `getRank(val)`, `remove(index)`. Using *Decorator* based design, this should be the last added feature in order to allow these operations to be visible and accessible. Figure 5 shows some decoration classes as they were defined in the *Decorator* based design.

But these constraints are eliminated if *MixDecorator* pattern is used. Using this new design we can also add `Unique` after `Ranked` and all additional operations brought by `Ranked` are still available. *MixDecorator* assures the fact that all the additional operations are accessible even if the decoration that defined them is not the last.

The modification that we had to apply to the first design was to add a new decorator interface `IFeaturesOperations` derived from `IStorage` and a new class `FeaturesOperations` derived from `SpecializedContainer`, which defines all the additional operations that the features introduce. The concrete decorations features are derived now from the class `FeaturesOperations`, and not directly from `SpecializedContainer` (Figure 6).

4.3. **Framework design in more detail.** We have considered that: memory representation, iterability, and searchability are implicit properties of each collection type of the framework. Based on these, the `IStorage` interface is defined as is shown in Figure 7.

The class `SpecializedContainer` defines template methods for the methods of `IStorage`. These template methods call some *proxy methods* (initially declared in `Storage<T>` – Figure 4) that precede and succeed the calls of the actual storage methods. Some dependences could be defined on them (e.g. `prev_add()` and `post_add()` are used by all operations that implies insertion or setting of new values). When we work with a container with several decorations, the proxy methods of the decoration are called in a chain.

The symmetric features modify the specifications of some basic storage operations. Prefix proxy operations could modify the preconditions, and suffix proxy operations could modify the postconditions.

A decoration that corresponds to a feature that restrains the basic set of features, implements prefix proxy operations that block the execution of the operations that have to be excluded.

For example, `Unique` defines a decoration that assures that no duplicates are included into the container. More precisely it defines the method `prev_add` in such a way that if the element is already into the container, the add operation of the storage support is no longer called.

Also, since for the collections, the decorations are dependent on each other, the proxy operations are called also in the recursive process specific to *MixDecorator* – the process of finding the concrete definition of the method. When methods corresponding to features are defined in the decorator class, besides forwarding the call to the base, prefix and suffix operations are called, too.

4.4. **Framework usage.** As we have specified before, the framework allows the creation of new collections by adding the characteristic features. We will give few examples.
If the user needs a stack of integers, with unique value elements he/she may use the following code:

```
1 FeaturesOperations<Integer> stack = new Unique(new Stack<Integer>(new Array<Integer>()));
2 //FeaturesOperations<Integer> stack = new Stack<Integer>(new Unique(
3 //                                        new Array<Integer>()));  equivalent
4     stack.add(new Integer(2));
5     Integer i = stack.extract();
```

If after using the stack for the initial purpose, based on the LIFO principle, the stack is not empty, the same storage could be used as a simple collection:

```
1     IStorage<Integer> support = stack.getStorage();
```

If for elements which are remained in the collection we would like to have direct access based on the rank, we may add `Ranked` feature:

```
1     Ranked<Integer> rank_coll = new Ranked<Integer> (support);
2     for (int i =0 ; i<rank_coll.size(); i++){
3         System.out.println("the␣next␣element␣in␣the␣stack"+ rank_coll.getElem(i));
```

5. Conclusions and Future Work

The proposed pattern – *MixDecorator* – is similar to *Decorator* pattern in the sense that allows functionality extension, but it brings an important advantage by allowing new responsibilities to be added. It treats 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). Different combinations of these messages could be used, and all the responsibilities are directly accesible.

Other important advantages are provided:

- `IComponent` is independent of any decoration declaration, so already defined concrete classes could be used;
- it allows a multi-stage development, and so extensions are possible and they benefit of the same advantages as the first defined set of decorations.

The implementation constraint of the solution that allows future decoration extensions is related to the fact that we have to be able to add a set of operations to an interface and also to provide a basic implementation for the corresponding methods. This could be achieved by using multiple inheritance, a surrogate of it, as that provided by the Java extended interfaces, or other mechanisms specific to the implementation language – as C# extension methods.

As the example with the data structures emphasizes, the applicability of the *MixDecoration* pattern is clearly defined and brings important advantages over the classical one. The pattern has been used in order to allow the creation of new collections based on dynamic composition of the features that characterize the corresponding data structures. In this way we may add or remove features dynamically. The fact that only linear combinations of features are allowed could be seen as a disadvantage, but since the classical version of *Decorator* has been replaced with *MixDecorator* this disadvantage is hidden: the features with changed interface are visible even if they are not added as a final decoration. Here the advantage of using the presented pattern is very clear. Without it the framework design and usage would have been much more difficult.

Other, more classical, examples could be given (e.g IO streams or graphical windows).

Since *MixDecorator* applicability is related to the possibility of adding new functionalities, a comparison with mixins could be done: MixIn programming is a style of software development where units of functionality are created in a class and then mixed in with other classes [BC90]. A mixin class could be considered as a parent class that is inherited from - but this is not done in order to obtain a specialization. Typically, the mixin will export services to a child class, but no semantics will be implied about the child "being a kind of" the parent.

The main differences between *MixDecorator* and Mixins are based on the fact that with *MixDecorator* we want to add functionality to objects, not to create new classes that contains a combination of methods from other

classes. With *MixDecorator* we may extend functionality (change behavior and add new responsibilities) of an object, and this new functionality could be added and removed dynamically.

As further work we will try to investigate more examples where using *MixDecorator* could bring important advantages, and also to investigate its variants.

Acknowledgement The author is grateful to EuroPLoP shepherd Eden Burton, and to the workshop participants for their constructive and helpful feedback and comments.

References

[BG97] [BG97] D. Batory,B.J. Geraci:*Composition Validation and Subjectivity in GenVoca Generators*. IEEE Trans. Software Engineering'97.

[BC90] [BC90] G. Bracha, W. Cook. 1990. *Mixin-based inheritance*. In Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications (OOPSLA/ECOOP '90). ACM, New York, NY, USA, 303-311.

[BMN14] [BMN14] 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.

[CE00] [CE00] K. Czarnecki, U. Eisenecker: *Generative Programming*. Addison Wesley, 2000.

[GHJV94] [GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object Oriented Software*, Addison-Wesley, 1994.

[Nic11] [Nic11] V. Niculescu,: *Storage Independence in Data Structures Implementation*, Studia Universitatis "Babes-Bolyai", Informatica, Special Issue, LVI(3), pp. 21-26, 2011.

[NL13] [NL13] V. Niculescu, D. Lupsa: *A Decorator based Design for Collections*.Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT2013, Cluj-Napoca (Romania), July 5-7, 2013, pp 54-64.

[ST04] [ST04] A. Shalloway, J. R. Trott. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. Addison Wesley, 2004

[Java8] [Java8] *Java SE 8: Implementing Default Methods in Interfaces*
http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/ JavaSE8DefaultMethods/JavaSE8DefaultMethods.html

[CS] [CS]*Extension Methods (C# Programming Guide)* https://msdn.microsoft.com/en-us//library/bb383977.aspx