# Implementations of the Decorator Pattern conceived for Alternative Presentations of the Object-Oriented Programming Paradigm

Nicolás Passerini[1][2][3] and Carlos Lombardi[1]

[1] Universidad Nacional de Quilmes, Bernal, Argentina
{npasserini,carlombardi}@gmail.com
[2] Universidad Tecnológica Nacional, Facultad Regional Buenos Aires, Argentina
[3] Universidad Nacional de San Martín, Miguelete, Argentina

**Abstract.** The software industry has identified, in a vast proportion, the object-oriented programming paradigm with the variant based in classes and single inheritance. Particularly, *design patterns* and their best-known implementations are largely conceived for this traditional object model. However, the last years have witnessed a steady growth of the popularity of several alternative models, which allow for different ways to organise the definition of object behavior.
This paper shows non-standard implementations of the decorator design pattern, which exploit behavior sharing and refining features unique to alternative models. We claim that these features allow for more natural renderings of the pattern and avoid a well-known flaw of the classical implementation, i.e. the loss of object identity.

## 1 Introduction

In the evolution of object-oriented programming, several languages have been developed implementing different variants of the paradigm. We can distinguish *class-based* languages, such as Smalltalk [6], Java or C++ from *object-based* languages, such as Self [12] or Javascript. Moreover, different *behavior sharing* mechanisms have been proposed for class-based languages, such as single inheritance, multiple [3] inheritance, mixin-based inheritance and traits.

For many years one of these variants has been by far more popular than the others. This variant is strongly class-based: every object is an instance of a class, an object cannot change its class once created, and the definition of behavior of objects lies entirely in classes. *Inheritance* is the only way to share behavior between classes without explicit coding. Besides, it is restricted to *single* inheritance. In the sequel, we will refer to this model as the *traditional* model of the object-oriented paradigm.

However, in the last few years, several alternative models have increased their popularity. Many languages have incorporated alternative ways of sharing behavior, such as Ruby, Groovy, Pharo Smalltalk, Scala[9]. Some of them allow you to add behavior directly to objects instead of classes, while others allow for different inheritance mechanisms. The different behavior sharing features included in these languages, which imply a change from the traditional model of object-oriented programming, are increasingly adopted in industrial developments.

Many of the technological, conceptual, and methodological tools that are most widely used in the software industry are conceived for languages following the traditional object model. The exploit of alternative behavior sharing mechanisms is sometimes hindered by the lack of tools needed for a medium or big scale development. For example, the description and classical implementations of design patterns [5] is given almost exclusively from the viewpoint of the traditional object model. Their applicability to alternative models is largely unexplored. Particularly, it is not clear if different behavior sharing facilities could give rise to better implementations of classical patterns.

In this work we describe two alternative implementations of the concept of *decorator* object. The proposed implementations take profit of ways to organise, share and refine the definition of object behavior which are not present in the traditional object model. Namely, we use *mixin-based inheritance* in one of the given implementations, and *delegation-based inheritance* in the other one.

We claim that the novel behavior sharing features allow for more natural renderings of the pattern. Moreover, the proposed implementations solve some negative consequences of the classical rendering of the decorator pattern. Particularly, object identity of the decorated component is preserved, as will be explained in the sequel.

In our opinion, a similar approach is worth taking for other popular design patterns; similar and even better (by comparison to classical) implementations could be obtained by exploiting different ways of organising object behavior definitions. This effort could contribute to provide the support needed in order to forward the adoption of alternative object models in software industry.

**Related work** The first systematic description of design patterns [5] gave rise to a broad literature about the subject discussing the applicability and consequences of patterns, proposing additional patterns, describing implementations tailored for different programming languages, etc. [1, 4]. There are a few later presentations that have focused in a programming language with an alternative model, such as AspectJ [7] or Scala [8]. To our knowledge, there is no study comparing pattern implementations among several alternative models.

**Structure of the paper** In Section 2 we briefly describe the traditional object model and the alternative behavior organisation and sharing features we will study in the sequel; and we introduce the programming languages which will be used in the examples. In Section 3 we describe and discuss the proposed, alternative renderings of the decorator design pattern. Finally, in Section 4 we present our conclusions and suggest further work.

**Acknowledgments** To Esteban Lorenzano for extensive discussions about the subject, and to Guillermo Polito, Mariano Martínez Peck, Leonardo Gassman, Javier Fernandes, Pablo Tesone, Débora Fortini and Ana María Larramendy for the review of preliminar versions.

## 2 Object-Oriented Models

In this section we will briefly describe three variants of the object-oriented paradigm, namely: the traditional model, based in classes and single inheritance; a model which allows for mixin-based composition; and finally an object-based model with delegation-based inheritance. The programming languages Scala and Ioke will be used to illustrate, respectively, the first two models and the last one; some features of them will be described in this section as well.

The focus will be set in the mechanisms provided in each model to organise, share and refine the definition of object behavior, and also in *method lookup*, i.e. the selection of the method to evaluate when an object receives a message.

### 2.1 Single Inheritance in Class-Based Languages

Class-based languages with single inheritance represent the most tratidional model of the object-oriented paradigm and by far the most popular one. Java, Smalltalk and C# are[4] some well-known representatives of this model. In this view *classes* play a fundamental role: every object is an instance of a class and an object cannot change its class once created. The methods, which give the definition of the behavior of objects, lie completely inside classes; this model does not provide mechanisms to attach behavior to individual objects.

*Single inheritance* is the only feature which allows for the sharing of behavior (i.e. method) definitions between different classes, allowing to refine in one class definitions given in other ones as well. Since each class has a unique superclass[5], traversing the superclass relationship recursively gives a list of superclasses, named the *ancestors* of a class.

The behavior of an object is defined by its class and its ancestors. When an object receives a message, method lookup starts in the class of the receiver object, and then traverses the list of ancestors in order until a suitable method is found. This resolution of method lookup allows a class, at the same time, to *inherit* (i.e. share) the method definitions from its ancestors, to change the behavior corresponding to a given message by *overriding* (i.e. providing a new definition for) the corresponding inherited method, and to extend the set of messages by defining *new* methods (i.e. methods that do not override any of the methods inherited from the ancestors).

Additionally, the use of the keyword `super` inside an overriding method, makes it possible to include an invocation to the overriden method, and therefore to mingle the inherited behavior within the new code. This feature extends the possibility to refine, in a class, the definitions inherited from its ancestors.

Within the execution of the method corresponding to a message-send, the *object identity* of the receiver is preserved. This allows most object-oriented languages to provide a way to send, within a method, additional messages to the

[4] At least in the ways in which these languages are most commonly used.

[5] This is the difference with multiple inheritance, in which a class can have several direct superclasses.

receiver object; a *pseudo-variable* (usually named `self` or `this`) is provided, which refers to the receiver inside a method body. In the so-called *self-sends*, the method lookup starts in the class of the receiver, no matter where the definition of the executing method lies. This is a very important characteristic on inheritance-based object-oriented systems because it permits some complex interaction patterns, such as *Template Method* [5, p. 325].

### 2.2 Mixin-Based Inheritance in Scala

The object model implemented in the Scala language extends the traditional model by incorporating *mixins*. A mixin [2] is a unit of definition of object behavior which includes method definitions. While any object is still instance of a class, a list of mixins can be incorporated into an object when it is created. Therefore, mixins provide an additional mechanism to share behavior definitions among objects, which furthermore can pertain to different classes. Thus, the definitional units provided by mixins are highly reusable and give rise to a compositional way of defining object behavior, known as *mixin-based composition*.

Figure 1 shows an example of mixin-based composition in Scala. `AbsIterator` defines a simple interface for iterators, which is implemented by `StringIterator`. `RichIterator` is a mixin that can be applied to any subclass of `AbsIterator` and extends its interface by defining the `foreach` method. Note that the `foreach` method uses `hasNext` and `next`, defined by `AbsIterator`. The example shows the creation of an object which adds the characteristics defined in `RichIterator` to those defined in `StringIterator`. Mixins are defined by the keyword `trait`[6].

```
abstract class AbsIterator[T] {
  def hasNext: Boolean
  def next: T
}

trait RichIterator[T] extends AbsIterator[T] {
  def foreach(f: T => Unit) { while (this.hasNext) f(this.next) }
}

class StringIterator(s: String) extends AbsIterator[Char] {
  private var i = 0
  def hasNext:Boolean = i < s.length()
  def next:T = { val ch = s.charAt(i); i += 1; return ch }
}

val iter = new StringIterator("Hello World") with RichIterator
iter.foreach(e => println(e))
```

**Fig. 1.** Mixin-based inheritance example in Scala

[6] The behavior of Scala `traits` resembles more tightly the idea of mixin, as defined by [2] than the idea of *trait* as defined by [11]. The Scala literature uses both words.

Mixins can be incorporated to objects, classes or other mixins. Thus, in this model the ancestors of an object do not conform a list, but an acyclic graph. In order to do method lookup, the graph of ancestors is *linearised*[7], i.e. the language semantics defines an order in which the graph has to be traversed. Particularly, method lookup gives, to mixins added to the message receiver, precedence over its class[8]. The ability to override methods, and also to access to the overriden versions by using the `super` keyword, is given to mixins analogously as described for classes. We remark that a `super`-send included in a mixin can refer to different overriden methods, depending on the particular configuration of each object incorporating the mixin.

## 2.3 Delegation-Based Inheritance in Ioke

The Ioke[9] language implements an object-based model, whose most prominent feature is the absence of classes. Thus, object behavior is defined inside objects.

Ioke implements the *delegation* mechanism of behavior definition sharing[10] which allows for the delegation of behavior between objects with no specific implementation code required. A list of *parents* is included in the definition of each object. When an object receives a message, the method lookup starts with the receiving object, and continues (if needed) with the list of its parents, traversed in order. Therefore, definitions included in an object are inherited by (i.e. shared with) all other objects including the former in their lists of parents. As described for mixin-based inheritance in Section 2.2, the behavior of an object is defined in a compositional manner; in this case, each object incorporates the behavior defined in all of its parents. Another similiarity is that the linearisation semantics is also used in Ioke[11]. The comment of Section 2.2 about overriding methods and the `super` keyword are valid as well.

This model preserves object identity. If an object executes a method inherited from one of its parents and this method uses the `self` pseudo-variable, it will refer to the original receiver, and not to the method owner. Therefore, delegation-based inheritance has at least the same expresive power as class-based inheritance. A remarkable advantage is that the set of parents of an object can be changed dynamically. Thus, dynamical changes to the behavior of an object are much more natural than in class-based languages.

Figure 2 shows an example of delegation-based inheritance in Ioke. In Ioke parents are named *mimics*. Sending the message `mimic` to any object creates a new object that has the receiver as parent. The message `do(...)` allows to add new features to the receiver and returns it. The messages `mimic! /`

[7] In other objects models the method lookup is based on the *flattening*, rather than the linearisation, of the ancestor graph.

[8] The details of method lookup and linearisation in Scala can be found in [9].

[9] For more information about Ioke, see `http://ioke.org/wiki/index.php/Guide`

[10] Delegation, along with cloning, are the best-known ways to organise and share behavior definitions in object-based languages

[11] This is the main reason because we choose Ioke over other interesting object-based models, such as [12] or [10]

```
StringIterator = Origin mimic do(
  initialize = method(aString, self i = 0. self s = aString)
  hasNext = method(i < s length)
  next = method(ch = s[i]. self i += 1. ch))

RichIterator = Origin mimic do(
  foreach = method(f, while(hasNext, f(next))))

iter = StringIterator mimic("Hello World")
iter mimic!(RichIterator)
iter foreach(fn(elem, elem println))
```

**Fig. 2.** Delegation-based inheritance example in Ioke

`prependMimic!` allow to add a new parent to the receiver, at the end / beginning of the parent list respectively.

## 3 Decorator

The intent of the *Decorator* pattern is to add responsibilities to individual objects, instead of adding them to an entire class [5, p. 175]. Decorators provide a flexible alternative to subclassing for extending functionality.

The classical implementation of this pattern is based on the traditional model of object orientation. To add responsibilites to a single object, we define an additional object which exhibits the same interface. The additional object will serve as a *decorator*: it implements the added behavior and delegates the rest in the original (or *decorated*) object. Since the decorator shares its interface with the decorated object, it can be furtherly decorated in turn, making it to define objects consisting of a basic implementation object combined with multiple decorators. Figure 3 shows an example of the classical implementation, in which two decorators are applied to a `CollectionStream`: a `Capitalize` that changes the first letter to upper case, and an `OnlyLetters` that discards any non-alphabetic characters. The abstract class `Stream` defines the common interface to decorators and decorated objects. Finally, `BaseDecorator` is an utility class that simplifies the implementation of concrete decorators, by delegating all necessary messages to the decorated object.

The classical implementation has three drawbacks. In the first place, decorator and decorated are different objects. Therefore, when a reference to `this` appears in any of the involved classes, it is not clear whether the referred object is the one expected.

We can show a concrete consequence in the example. The result of the evaluation of `s1.writeAll("hello34")`, will include the numbers, because the method `write` in the `OnlyLetters` decorator (which would have rejected them) is never invoked. The implementation of `writeAll` is delegated, from `OnlyLetters` to `Capitalize` with no particular action (since `OnlyLetters` only overrides the `write` method), and then to `CollectionStream` after having capitalized the

```
abstract class Stream {
  def write(elem: Char): Unit
  def writeAll(elems: String): Unit
}

class CollectionStream extends Stream {
  var data = new MutableList[Char]();
  def write(elem: Char): Unit = { data += elem }
  def writeAll(elems: String) = elems.foreach(e => this.write(e))
}

abstract class BaseDecorator(deleg: Stream) extends Stream {
  def write(elem: Char): Unit = deleg.write(elem)
  def writeAll(elems: String): Unit = deleg.writeAll(elems)
}

class Capitalize(deleg: Stream) extends BaseDecorator(deleg) {
  override def writeAll(elems: String) = {
    this.write(elems.head.toUpper);
    deleg.writeAll(elems.tail);
  }
}

class OnlyLetters(deleg: Stream) extends BaseDecorator(deleg) {
  override def write(el: Char) = if (el.isLetter) deleg.write(el)
}

val s1 = new OnlyLetters(new Capitalize(new CollectionStream))
```

**Fig. 3.** Traditional implementation of the decorator pattern

first letter. When the method `writeAll` in `CollectionStream` is executed, the object referred to by `this`, i.e. the *receiver*, is the original stream, which is a different object from any of the decorators. Therefore, the method lookup for the `this.write` send will not include methods defined in decorators.

Notice that this unwanted behavior could be fixed by moving the implementation of `writeAll` to the `Stream` class, and deleting the corresponding implementation in `BaseDecorator`. Now the code is executed with the `OnlyLetters` decorator as receiver, and then method lookup for `this.write` will begin with that class. Unfortunately, the decorated `writeAll` implementation in `Capitalize` is ignored, and therefore the first letter will not be capitalized as expected.

The second drawback is that delegation to the decorated object has to be explicitly coded, in the `BaseDecorator` in the example.

Finally, we remark that there is no easy way to *dynamically* add responsibilites to an object using this implementation of the pattern. Adding a decorator to an object implies that subsequent messages must be sent to the new decorator in order to access the added behavior. Therefore, all references to the original object should switch to the decorator, a task being unfeasible or at least problematic in most object-oriented environments.

**Mixin-based implementation** Mixin-based inheritance allows for a different approach to the decorator pattern: implement each decorator as a mixin. Figure 4 shows a mixin-based implementation of the decorator pattern. `Stream` and `CollectionStream` remain the same as in the traditional implementation, while `BaseDecorator` is no longer needed.

```
trait Capitalize extends Stream {
  abstract override def writeAll(elems: String) = {
    this.write(elems.head.toUpper);
    super.writeAll(elems.tail);
  }
}

trait OnlyLetters extends Stream {
  abstract override def write(el: Char) = if (el.isLetter)
      super.write(el)
}

val s1 = new CollectionStream with Capitalize with OnlyLetters
```

**Fig. 4.** Mixin-based implementation of the decorator pattern in Scala

Unlike [8][12], we find that the mixin-based implementation solves two of the problems in classical decorator implementations.

In the first place, the original object incorporates the added behaviors, instead of being decorated by additional objects. Therefore, object identity is preserved, and consequently the object will behave as expected in all situations. In the example, `s1` incorporates the definitions included in `OnlyLetters`, `Capitalize` and `CollectionStream`; these units of behavior definition are included in the linearised list of ancestors in the given order. If we evaluate `s1.writeAll("hello34")`, then the method `writeAll` of `Capitalize` is executed; notice that `this` refers to `s1`. Moreover, the method lookup for the `this.write` call begins with `OnlyLetters`.

In the second place, delegation is achieved avoiding the need of manually written code, taking advantage of the method lookup mechanism.

**Delegation-based implementation** Still another rendering of the decorator pattern can be built by using delegation-based behavior sharing, cfr. Section 2.3. The corresponding implementation is shown in Figure 5. A decorator is implemented as an object which is added as mimic to the original object. The use of `prependMimic!` allows the decorator to be the first receiver of the messages.

We remark that in this implementation, decorators can be added or removed at any moment without changing the identity of the object. Moreover, the addition of decorators as mimics does not alter the object identity of the decorated object: the mimics lie “behind” the mimicked object. The `super` keyword allows a decorator to refer to the previous object w.r.t. the linearisation semantics

[12] whose conclusion is that “Scala does not offer something new with respect to the Decorator pattern”

```
CollectionStream = Stream mimic do(
  initialize = method(self data = [])
  write = method(elem, data append!(elem))
  writeAll = method(elems, elems each(elem, write(elem))))

Capitalize = Origin mimic do(
  writeAll = method(elems, write(elems head upper)
                           super(elems tail)))

OnlyLetters = Origin mimic do(
  write = method(elem, if(elem isLetter, super(elem))))

s1 = CollectionStream mimic do(prependMimic!(Capitalize)
                               prependMimic!(OnlyLetters))
```

**Fig. 5.** Delegation-based inheritance implementation of the decorator pattern

of mimics, thus giving decorators the ability of invoking the overrided behavior. Therefore, method lookup works in this example analogously to what we have described for the mixin-based implementation. Finally, we notice that this method lookup also avoids the need to write explicit delegation code. Therefore, all the drawbacks described for the classical implementation are avoided.

## 4 Conclusions and Further Work

In this work we have analysed some possible implementations of the decorator patterns using alternative variants of the object-oriented paradigm. Exploiting the possibilities to share and refine behavior present in the alternative variants, we found implementations of the decorator pattern that are easier to code and maintain than the classical implementation, and also allow forms of use that would be more difficult to obtain using the traditional variant of the object-oriented paradigm.

Not surprisignly, mixins are much more flexible than single inheritance. In a single-inheritance model, a subclass represents a slice of behavior that is added to a specific superclass. On the other hand, a mixin represents a slice of behavior that can be applied to an individual object. Therefore, mixins proved to be a very useful tool to define small behavioral units that can be later composed in multiple ways. Modelling decorators as mixins preserves object identity, in the sense that all references to `self` in the code of all behavioral units used to build the decorator will point to the same object.

Delegation-based inheritance outweighs the advantages of mixin-based inheritance, by defining sharable behavior as individual objects rather than as classes or mixins. The most visible advantage of this kind of inheritance is the possibility of dynamically changing the behavior of an object. The decorator example shows that, in some situations, delegation-based inheritance simplifies dynamic changes to the behavior of an object, which would be more complicated in the traditional view of object-oriented programming.

In both alternative models analysed in this work the preservation of the `self` variable plays a central role. One of the design principles defined by Gamma *et. al.* states that, in order to improve object oriented design, you should sometimes *replace inheritance by delegation*. Still, in the traditional view of object-oriented programming, delegation has to be done manually and the `self` variable is not preserved. Our example shows that a behavior sharing mechanism that preserves the identity of `self` serves as a basis for more flexible combinations of behavioral units. Each time that in the definition of the behavior of an object a message `m` is sent to `self`, both analized alternatives makes it possible to compose the definition of that object with an additional behavioral unit, which overrides exactly the behavior asociated with `m` without requiring any change to the original behavioral units. This way of overriding behavior can be seen as an extended version of the *Template Method* design pattern [5].

In future work, we plan to extend the kind of analysis performed in this article to other design ideas, including other design patterns. We think that this may contribute to the creation of a conceptual framework that fosters the explotation, in the software industry, of the advantages provided by the alternative models of the object-oriented paradigm.

## References

[1] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The design patterns Smalltalk companion.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[2] Gilad Bracha and William Cook. Mixin-based inheritance. *SIGPLAN Not.*, 25(10):303–311, September 1990.

[3] Luca Cardelli. A semantics of multiple inheritance. In *Proc. of the international symposium on Semantics of data types*, pages 51–67, New York, NY, USA, 1984. Springer-Verlag New York, Inc.

[4] James W. Cooper. *The Design Patterns Java Companion.* 1998.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[6] A. Goldberg and D. Robson. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc., 1983.

[7] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, November 2002.

[8] F.S. Løkke. *Scala and Design Patterns: Exploring Language Expressivity.* Aarhus Universitet, Datalogisk Institut, 2009.

[9] Martin Odersky. The Scala language specification. 2009.

[10] J. Ressia, T. Gırba, O. Nierstrasz, F. Perin, and L. Renggli. Talents: Dynamically composable units of reuse. *IWST 2011*, page 109.

[11] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. In *ECOOP*, pages 248–274, 2003.

[12] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM.