Classes as Layers: Rewriting Design Patterns with COP

Alternative Implementations of Decorator, Observer, and Visitor

Matthias Springer[‡] Hidehiko Masuhara[‡] Robert Hirschfeld[†,§]

[‡] Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Japan
[†] Hasso Plattner Institute, University of Potsdam, Germany
[§] Communications Design Group (CDG), SAP Labs, USA; Viewpoints Research Institute, USA
matthias.springer@acm.org masuhara@acm.org robert.hirschfeld@hpi.de

Abstract

This paper analyzes and presents alternative implementations of three well-known Gang of Four design patterns: Decorator, Observer, and Visitor. These implementations are more than mere refactorings and take advantage of a variant of context-oriented programming that unifies classes and layers to overcome shortcomings in a conventional, object-oriented implementation.

Keywords Design Patterns, Decorator, Observer, Visitor, Context-oriented Programming, Layers

1. Introduction

Software design patterns are reusable blueprints for problems that occur frequently in software design. Among them are the most prominent ones by "Gang of Four" [2]. We selected three of their design patterns (Decorator, Observer, and Visitor), identified shortcomings in a conventional object-oriented implementation, and present an alternative implementation using layer-based context-oriented programming (COP). These new COP implementations are not merely refactorings of their OO counterparts; Decorator and Observer differ in their semantics and solve functional inadequacies, in addition to making the source code in our opinion more understandable by reducing object interaction.

The new implementations are based on a new conceptual idea for organizing partial methods. In most COP frameworks, partial methods belong to a layer. In our system, there is no dedicated layer construct and partial methods belong to classes, allowing arbitrary objects to affect other objects by intercepting (layering) their methods. The following sections will give an overview of the COP mechanism used in this paper and present the alternative implementations of the three design pattern using that mechanism. For every design pattern, we present an example, identify shortcomings in a conventional implementation, show a COP-based implementation, and discuss consequences and possible disadvantages of our solution.

COP'16, July 17-22 2016, Rome, Italy

ACM 978-1-4503-4440-1/16/07...$15.00. http://dx.doi.org/10.1145/2951965.2951968

2. Mechanism and Notation

For the implementation of the design patterns that we show in Section 3, a number of features and mechanisms are required that are not typically found in COP frameworks. We evaluated some of these features separately in our previous work but not together in this combination. The code listings shown in the remainder of this paper are written in an imaginary dynamically-typed, class-based, object-oriented programming language with Java-like syntax.

Partial Methods In layer-based context-oriented programming [4], layers can refine methods of other classes. Such refinements are called *partial methods*. When a layer is active, the method lookup first looks for a corresponding partial method in that layer, before falling back to the original implmentation. Multiple layers can be active at the same time, forming a *layer composition stack*.

Classes as Layers This work builds on top of ideas from our previous work which unified layers and classes [7]. We assume that there is no dedicated layer construct. Instead, classes and their instances can act as layers. In addition to member methods and static methods, classes can provide member partial methods and static partial methods. A partial method can refine existing methods or add new methods to the *target class*.

Since layers are implemented by objects, arbitrary (layer) objects can be activated/deactivated. We use the term *layer object* to denote an object that acts as a layer in a certain situation and *affected object* to denote the object whose behavior is adapted. As long as a layer object is active (see paragraph *Layer Activation*), its partial methods are in effect. If the layer object is a class, its static partial methods are

in effect. If the layer object is a non-class object, its class's member partial methods are in effect.

Accommodating State Layer objects can refine existing and add new methods to affected objects (via partial methods), but they cannot add additional instance variables. There are three reasons for this decision: First, it is unclear how conflicts should be handled, i.e., what if two layer objects define an instance variable with the same name. Second, adding new instance variables is difficult from an implementation point of view, because they would have to be stored separate from the other instance variables. Third, such functionality is not required for the examples in this paper. Instead, a layer object's class can define its own instance variables that can be accessed inside partial methods. If there is a one-to-one mapping between a layer object and an affected object, this is as if the instance variable belonged to the affected object.

Partial methods effectively belong to the target class, i.e., a class different from the layer class. Method calls and instance variable references inside a partial method are by default resolved in the target class, but we assume that the `thisLayer` keyword can be used to access instance variables of the layer class.

- Keyword `thisObject`[1]: Lookup in the affected object
- Keyword `thisLayer`: Lookup in the layer object
- Keyword `this`: Try lookup in affected object, then layer object
- No keyword given: same as `this`

The following listing shows the `this` and `thisLayer` in a simple example with a member method `foo` and a member partial method `bar`.

```
class A { // target class
   def a;
   def b;
}

class B { // layer class
   def b;
   def c;

   def A.bar() { // partial method for A.bar
      this.a; // -> A.a
      this.b; // -> A.b
      this.c; // -> B.c
      thisLayer.a; // -> error
      thisLayer.b; // -> B.b
      thisLayer.c; // -> B.c
   }

   def foo() {
      this.a; // -> error
      this.b; // -> B.b
      this.c; // -> B.c
      thisLayer.a; // thisLayer undefined
      /* ... */
   }
}
```

[1] Not used in this paper, only mentioned for the sake of completeness.

If the affected object or layer object is a class, then `this` or `thisLayer` references static methods and fields. Otherwise, it references member fields.

Layer Object Activation We assume that layer objects can be activated globally, in a block scope, and for a certain affected object. In the first case, a layer object is activated by calling its `activate` method and remains active in the entire program until it is deactivated explicitly. In the second case, a layer object is activated using `with` statements taking the layer object as an argument and followed by a block, within which the layer object remains active. In the third case, a layer object is activated only for a single affected object by calling the affected object's `activate` method with the layer object as an argument, and the layer object remains active until it is deactivated explicitly. This is contrary to the first two cases, where more than one object might be affected (all instances of classes for which a partial method is defined).

```
def affectedObject = new A();
def layer = new B();

// global activation
layer.activate();
layer.deactivate();

// block scope activation
with (layer) { /* active in here */ }
without (layer) { /* inactive in here */ }

// per-object activation
affectedObject.activate(layer)
affectedObject.deactivate(layer)
```

There must be a mechanism in place to determine the precedence of these three activation mechanisms. For example, if a layer object is activated globally and then deactivated for a single affected object, that deactivation should have precedence over the previous layer object activation. The details do not matter in this work; previous work has proposed various mechanisms [6].

3. Design Patterns

This section presents three well-known design patterns [2]: Decorator, Visitor, and Observer. For every pattern, there is an example implementation using the COP mechanism described in Section 2.

3.1 Decorator

The Decorator design pattern is used to add responsibilities to an object at run-time. A Decorator is typically implemented as a wrapper object holding a reference to the original object. Multiple Decorators can be applied by wrapping the object multiple times.

One disadvantage of a wrapper implementation is that responsibilities cannot be added or removed dynamically without loss of object identity. After wrapping a Decorator around an object, the resulting wrapped object has an object identity different from the original object (*object schizophrenia* [5]). If that wrapped object should be used everywhere the original object is in use, it has to be replaced one by one.

COP Implementation A Decorator is a layer object with partial methods for modified or additional behavior and is activated only for the decorated (affected) object. A partial method can issue a `proceed` call to combine the new behavior with the original one. As soon as a Decorator object is activated, it is active wherever the (original) decorated object is used, solving the problem of object identity. In cases where a Decorator adds state to a decorated object, there should be exactly one decorated object per Decorator object, with the additional state as instance variables of the Decorator object (many-to-one relationship between layer objects and affected object).

Example Figure 1 shows the data structure for a computer game that is based on 2D grid representation. The game consists of a matrix of game fields. Every game field has references to its neighboring fields and methods for drawing the field and for handling incoming entities. For example, the player entity might enter the game field from a neighboring field. Game fields can be decorated with the `BurningFieldDecorator`, representing a field that is on fire and causes damage to entering entities. This Decorator also modifies the `draw` method to show a fire animation.

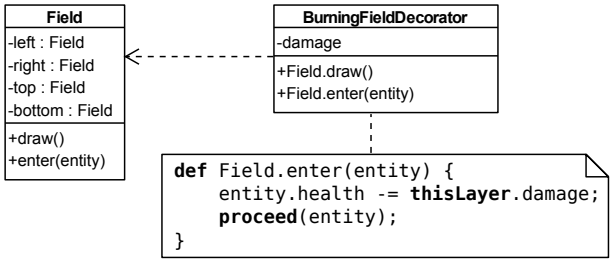


Figure 1: Example: Decorator Methods as Partial Methods

A field `field` can be set on fire by creating a decorator for it and applying it, as shown in the following listing.

```
def decorator = new BurningFieldDecorator();
decorator.damage = 15;
field.activate(decorator);
```

This example is hard to implement with a conventional Decorator, because the decorated object is referenced by its four neighboring fields. Whenever a field is wrapped with a Decorator, these references would have to be replaced by the decorated object. An alternative implementation could use object composition with a set of Decorator items (one of them being fire) for every field instead of a Decorator: Every field would act as a container for items and references would always point to fields instead of field wrappers. Field methods would delegate to item methods internally.

Consequences A COP Decorator implementation can differ from a conventional object-oriented implementation in a way that might not be anticipated by the programmer: Decorator methods are not only visible from the outside (when calling a method on the decorated object), but also from inside. This can be a problem if a method should behave differently in both cases. For example, a scroll bar Decorator might add the proportions of the scroll bar to the values returned by methods `width` and `height`, even if text rendering (implemented in the text box) depends on the original implementation.

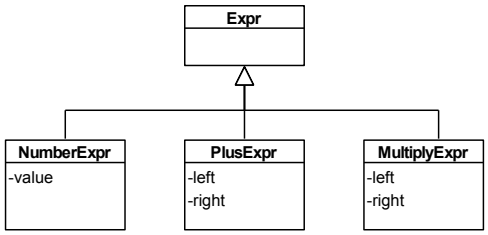
One limitation of a COP implementation of the Decorator design pattern is that the Decorator is always specific to a class, because COP relies on static types for the target class in the partial method signature even in dynamically-typed languages. For example, the Decorator in the example can decorate only `Field` objects but not other objects with the same method. The partial method could, however, target methods of an interface (or superclass) instead of a concrete class in statically-typed languages (e.g., the superclass `Object`).

3.2 Visitor

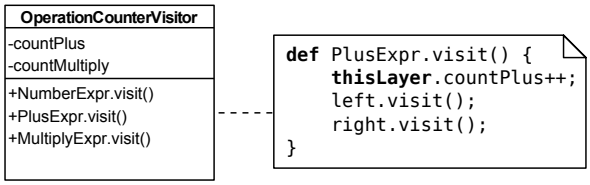
The Visitor design pattern is used to modularly add new operations to a set of classes, typically forming a tree structure, without having to modify these classes. Every class has an `accept` method taking a Visitor as an argument and calling a type-specific method on the Visitor in a double-dispatch fashion. The Visitor design pattern allows programmers to separate the base functionality of a set of classes from additional concerns that are only needed in certain situations and encapsulated in the Visitor class.

One disadvantage of a double-dispatch implementation is complex object interaction. Applying an operation provided by a Visitor to an object of unknown type requires calling an `accept` method on the object with the Visitor as an argument, which will then call the type-specific `visit` method.

COP Implementation A Visitor is a layer object with partial `visit` methods for all classes in the object structure. These methods do not refine existing base methods but are new methods. Within a `visit` method, it is possible to call the `visit` methods of associated objects directly and without double dispatch. Note, that `visit` methods can be renamed, making it possible to use multiple visitors at the same time, as long as there are no name clashes between visitor methods.



(a) Classes for Arithmetic Expressions



(b) Operations Counter Visitor

Figure 2: Example: Visitor Methods as Partial Methods

A Visitor can maintain internal state as instance variables of the Visitor object. In contrast to a conventional Visitor implementation, a COP-based Visitor is reduced to a container for partial methods and maybe internal state (i.e., the Visitor disappears and becomes part of the affected objects). In particular, there is no `accept` method anymore.

Example Figure 2(a) shows the class diagram for the tree node classes in an arithmetic expressions library. The library supports numeric literals, plus operations, and multiplication operations. Figure 2(b) shows the layer-based operation counter Visitor, which counts how often plus operations and multiplication operations appear in an expression. To that end, the layer object has two integer variables for counting these operations.

Before the Visitor can be used, it must be activated, for example using block-scoped layer activation. The Visitor can then be invoked by calling the `visit` method on the expression object. Inside a `visit` method, another subexpression can be visited by invoking its `visit` method without double dispatch.

```
def visitor = new OperationCounterVisitor();
with (visitor) {
  expression.visit();
}
println(visitor.countPlus + ", " + ↩
    visitor.countMultiply);
```

Note that per-object activation on `expression` is not sufficient, because the Visitor would then not be active for subexpressions.

Consequences In a traditional Visitor implementation, libraries often provide abstract Visitor classes with `visit` methods containing the object traversal code. A concrete Visitor would then overwrite `visit` methods with visiting code and a `super` call to visit associated objects. The following listing shows how an abstract Visitor can be implemented with COP.

```
abstract class ExpressionVisitor
  def PlusExpr.visit() {
    left.visit();
    right.visit();
  }

  /* ... /*
}

class OperationCounterVisitor extends ↩
    ExpressionVisitor {
  def PlusExpr.visit() {
    thisLayer.countPlus++;
    super.visit();
  }

  /* ... */
}
```

The interesting part of this example is the `super` call: Should this statement invoke the `visit` method in the superclass of `PlusExpr` or the partial `visit` method in the superclass of `OperationCounterVisitor`? In our previous work, we proposed the concept of an *effective superclass hierarchy* [7], i.e., a mechanism that formalizes the semantics of `super` calls. According to that mechanism, every class C in the superclass hierarchy is prepended with one (partial) class per active layer L, containing partial methods defined in L for that class C. Consequently, the `super` call in the example would call the partial method in `ExpressionVisitor`[2].

3.3 Observer

The Observer design pattern is used to deliver changes in the state of a *subject* to an *observer*. A subject typically maintains a list of observers and notifies them via their `update` method whenever its state changes. Information about the changed state is either directly passed as an argument to the `update` method or queried by the observer.

One shortcoming of this implementation is that there are no *levels* of notification. For example, some observers might only be interested in a certain kind of events, while others want to be notified about all state changes. However, a conventional implementation of the Observer design pattern does not let multiple observers specify when they should be notified[3].

COP Implementation An observer is a layer object with partial methods for the methods that trigger state changes.

[2] In the absence of other active layer objects, a `super` call in that partial method would try to lookup `visit` in `Expression` (starting with partial methods provided by layers), i.e. there is no `proceed` statement but the semantics of `super` is extended.

[3] The related Publish-subscribe pattern [1] does support this feature.

Such a partial method calls `proceed` and then triggers the change reaction code in the observer. The collection of observers and the code broadcasting change notifications to observers disappears and is hidden in the method lookup mechanism and `proceed` calls.

Example Figure 3 shows the class diagram for a user manager component. The user manager acts as the subject and supports checking credentials, creating new user accounts, and changing permissions for a given user. Two observers, a login monitor and a security log are defined. The login monitor is a live view for user activity and shows the number of successful and failed login attempts in a bar chart. The security log records more severe events such as account creation or changing user permissions, but not successful or failed login attempts. The crucial point in this example is that different events are relevant for the login monitor and the security log, which cannot be adequately handled in a conventional Observer implementation with only a single notification mechanism.

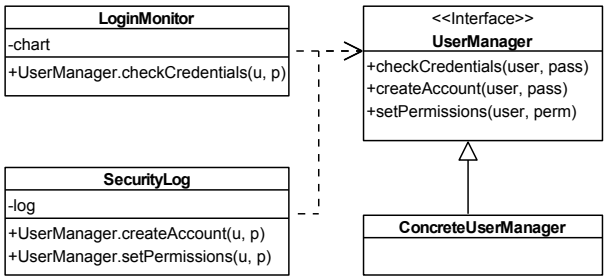


Figure 3: Example: Observer as Layer

Both observer classes define member partial methods for the respective methods in `UserManager`. The following listing shows an example for `LoginMonitor`.

```
class LoginMonitor {
 def UserManager.checkCredentials(u, p) {
  def result = proceed(u, p);
  if (result) {
   thisLayer.chart.bars["successful"]++;
  } else {
   thisLayer.chart.bars["failed"]++;
  }
  return result;
 }
}
```

Observers can be added (and removed) using layer activation statements. In the following listing, both observers are activated for a user manager instance `manager`.

```
def loginMonitor = new LoginMonitor();
def securityLog = new SecurityLog();
manager.activate(loginMonitor);
manager.activate(securityLog);
```

Consequences The example presented in the previous paragraph uses per-instance activation to observe a specific object. Global activation allows an observer to observe all instances of a class. For example, the login monitor could observe all user managers in a system if it is activated globally.

The partial methods of an observer should be confined to the public methods of the subject, as they would otherwise rely on internal implementation details and violate encapsulation. Moreover, in comparison to a conventional implementation, a COP-based implementation provides less flexibility as to when a change notification should be triggered. It is not possible to insert those notifications at an arbitrary point inside a method, because an observer can only listen to method call events. This limitation, in turn, might prevent entanglement of object responsibilities and change notifications.

Object-oriented Implementation Notification levels can be implemented without context-oriented programming using only object-oriented programming by providing multiple Observer interfaces. For every level, the subject maintains a separate list of observers and notifies only these observers about state changes corresponding to that level. In contrast to the COP approach, all notification levels have to be known at subject design time and cannot be introduced later.

4. Related Work

Previous work has analyzed the implementation of GoF design patterns in Java with AspectJ [3]. Contrary to this work, the authors do not change the design patterns themselves, but present a more modular and reusable implementation. The Observer design pattern is implemented with an abstract `Observer` aspect consisting of an abstract `subjectChange` pointcut, add/remove observer methods, a weak hash map for storing observers per subject, and advice for triggering observer updates. The authors propose AspectJ open classes for implementing the Visitor design pattern and attaching advice to implement the Decorator design pattern. These mechanisms are similar to our implementation approach, but context-oriented programming allows dynamic reordering of Decorators (using layer object (de)activation statements).

Previous work on *Reflective Designs* [5] shows how the Decorator design pattern and the Visitor design pattern can be implemented in AspectS. The authors propose using aspects and around advice to implement the Decorator pattern, which is identical to our approach. Decorators/proxies are instance-specific: aspects are instantiated and then activated for specific subjects (objects), which is similar to layer instances and our proposed mechanism for layer activation. Three different mechanisms are mentioned for implementing the Visitor design pattern: (a) an aspect providing the double-dispatch protocol, (b) one aspect containing all `visit` methods, and (c) a more generic implementation using traversal strategies. The second approach is similar to our approach, but it remains unclear whether such an aspect is an instantiation and how it is activated.

5. Summary and Future Work

In this paper, we presented alternative implementations of three well-known design patterns, based on an idea that unifies classes and layers. The Decorator implementation solves the problem of loss of object identity. The Observer implementation provides multiple notification levels and reduces object interaction. The Visitor implementation avoids double dispatch, also reducing object interaction. One limitation of COP-based implementations is that the method lookup does not distinguish between internal/external method calls (e.g., `this.foo` vs. `object.foo`), limiting opportunities for encapsulation.

Future work might analyze more than just these three design patterns, take into account real applications, and evaluate ideas for internal/external partial methods. This would allow a method to have different behavior, depending on whether it was called from code within the object or from external code. This would not only bring the Decorator implementation closer to its OO implementation (c.f. scroll bar example), but also allow for a COP implementation of the Adapter design pattern, which is hard to implement using COP if method name clashes exist between the original interface and the adapter interface.

References

[1] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003.

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

[3] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. OOPSLA '02, pages 161–173. ACM, 2002.

[4] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[5] Robert Hirschfeld and Ralf Lämmel. Reflective designs. *IEE Journal on Software, Special Issue on Reusable Software Libraries*, 152(1):38–51, Feb 2005.

[6] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Sci. Comput. Program.*, 76(12):1194–1209, December 2011.

[7] Matthias Springer, Hidehiko Masuhara, and Robert Hirschfeld. Hierarchical layer-based class extensions in Squeak/Smalltalk. MODULARITY Companion 2016, pages 107–112. ACM.