Classes as Layers: Rewriting Design Patterns with COP

Alternative Implementations of Decorator, Observer, and Visitor

Matthias Springer   Hidehiko Masuhara   Robert Hirschfeld

Dept. of Mathematical and Computing Science, Tokyo Institute of Technology
Hasso Plattner Institute, University of Potsdam

July 19, 2016

Overview

Introduction

- **Related Work:** Instantiable layers in JCop [1] etc., previous work on COP-based class extensions [2]
- **Idea:** Unify classes and layers; partial methods are defined as part of classes (i.e., classes can acts as layers)
- **This presentation:** How to rewrite Decorator, Observer, Visitor [3] to take advantage of that
  - Pattern description
  - Traditional implementation example
  - COP implementation example
  - Benefits and disadvantages
- Not mere refactorings, but rewritings: changed semantics

Overview

Language Design

- Classes can have 4 different kinds of methods:
  - Member method (instance method)
  - Member partial method (partial method defined for instances)
  - Static method (class method)
  - Static partial method (partial method defined for class)
- Arbitrary objects can be (de)activated (no dedicated layer construct)
  - Global activation
  - Block scope activation
  - Per-object activation [4]
- Object providing partial methods: *layer object*
- Object(s) being adapted: *affected object(s)*

Language Design

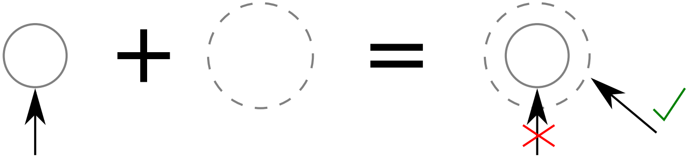
Example

```
class T {                                          /* target class */
  def foo() { /* ...  */ }
  def bar() { return "T"; }
}

class L {                                          /* layer class */
  def T.foo() {
    thisLayer.bar();                               /* -> "L" */
    this.bar();                                    /* -> "T" */
  }

  def bar() { return "L" }
}

new L().activate();                         /* global activation */
new L().activate(new T());               /* per-object activation */
with (new L()) { /* ...  */ }            /* block scope activation */
```

Overview

Decorator

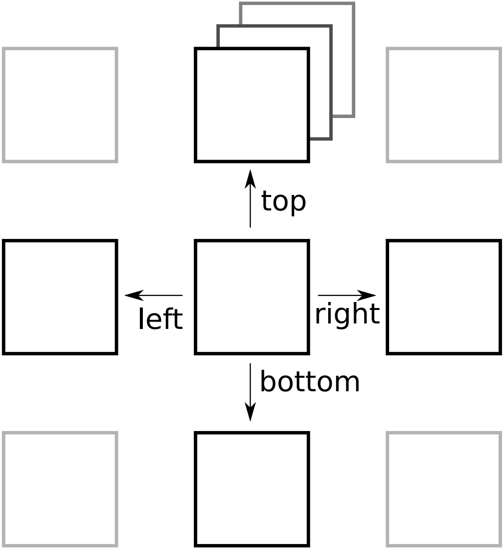
Pattern Description



- **Purpose:** Adding/removing responsibilities to an object at runtime
- **Mechanism:** Wrapping the object in a decorator, using the decorator instead of the object from now on
- **Problem:** References to the original object are not affected

Decorator

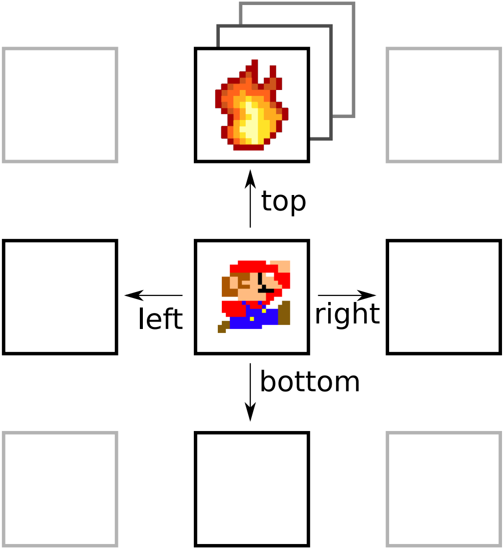
Example



- *Example:* Game with 2D grid (consisting of fields)
- Fields connected with adjacency lists
- Would like to ensure that references point to decorated fields

Decorator

Example



- *Example:* Game with 2D grid (consisting of fields)
- Fields connected with adjacency lists
- Would like to ensure that references point to decorated fields

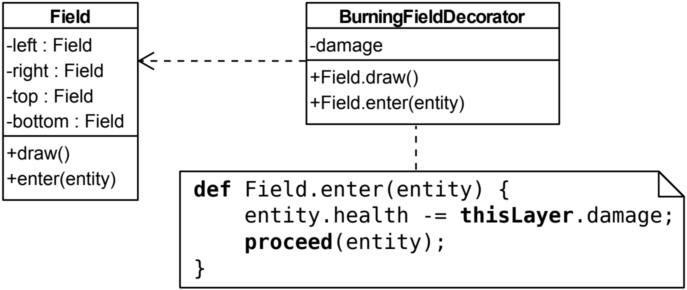
Decorator

Traditional Implementation: Example

```
class Field {
  def left, right, top, bottom;
  def draw() { /* ...  */ }
  def enter(entity) { /* ...  */}
  def neighbors() { /* ...  */ }
}

class BurningFieldDecorator {
  def decoratee;
  def damage = 15;

  def draw() { /* ...  */ }

  def enter(entity) {
    entity.health -= damage;
    decoratee.enter(entity);
  }

  def neighbors() { return decoratee.neighbors(); }
}
```

Decorator

COP Implementation: Example



- A decorator is an object that provides partial methods for additional/modified behavior
- Partial methods can call `proceed` to invoke next/original method

Decorator

COP Implementation: Example

```
def field = /* ... */
def decorator = new BurningFieldDecorator();

// Active decorator on object field
decorator.activate(field.left);

// Call decorated method
def moveLeft() {
  def player = /* ... */
  field.left.enter(player);
}
```

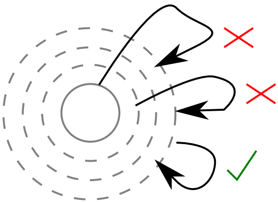
Decorator

COP Implementation: Example

```
def field = /* ...  */
def decorator = new BurningFieldDecorator();
def anotherDecorator = new MineFieldDecorator();

// Active decorator on object field
decorator.activate(field.left);
anotherDecorator.activate(field.left);

// Call decorated method
def moveLeft() {
  def player = /* ...  */
  field.left.enter(player);
}
```

Decorator

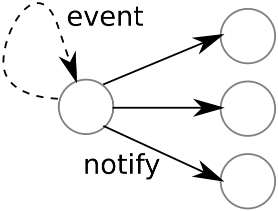
COP Implementation: Consequences



- ⊖ Method calls within an object (`this` calls) are affected
  Is that a bad thing if we layer only `public` methods?
- ⊖ Partial methods rely on static types for target class (i.e., `BurningFieldDecorator` can only layer `Field` objects)
  → Do we need wildcard class names? (`*.enter(entity)`)
- ⊕ No "object schizophrenia"

Observer

Pattern Description



- **Purpose:** Reacting to state changes/events of a dependent object
- **Mechanism:** Maintaining a list of observers, notifying all observers about state changes/events
- **Problem:** All observers are notified about all state changes/events
- **Problem:** Difficult to pass information about different events
- **Problem:** Troublesome to observe all instances of a class

Observer

Example

- Application with login, register functionality: class `UserManager`
- `LoginMonitor`: listens to login attempts
- `SecurityMonitor`: listens to failed login attempts and new user registrations

Observer

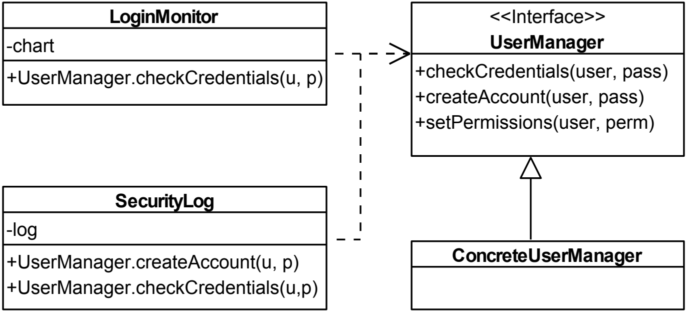
Traditional Implementation: Example

```
class UserManager {
  def observers = new List();
  def notify(type, data) {
    for (def o in observers) {
      o.update(type, data);
    }
  }

  def checkCredentials(user, pass) {
    notify("login", user)
    if (wrongPass) {
      notify("failed_login", user);
    }
  }

  def createAccount() {
    notify("create_acc", null);
  }
}
```

```
class SecurityLog {
  def update(type, data) {
    if (
      type == "failed_login" ||
      type == "create_acc") {
      /* ... */
    }
  }
}

class LoginMonitor {
  def update(type, data) {
    if (type == "login") {
      /* ... */
    }
  }
}
```

Observer

COP Implementation: Example



- An observer is an object that provides partial methods for methods indicating state changes/events
- Partial methods immediately call `proceed` and handle event

Observer

COP Implementation: Example

```
class SecurityLog
  def UserManager.checkCredentials(user, pass) {
    if (!proceed(user, pass)) {
      /* ...  */
    }
  }
}

def userManager = /* ...  */
def loginMonitor = new LoginMonitor();
def securityLog = new SecurityLog();

// Activate observer on object userManager
loginMonitor.activate(userManager);

// Activate observer on all UserManager implementation objects
securityLog.activate();
```

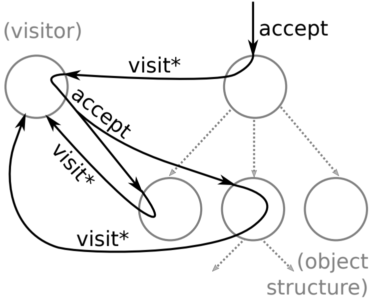
Observer

COP Implementation: Consequences

- ⊖ **Less Flexibility:** Notifications only before or after method calls, but not inside (less flexibility)
- ⊖ **Modularity:** Potentially tighter coupling between subject and observer (binding observer to method names of subjects)
- ⊕ **Argument Passing:** Every partial method can have its own signature
- ⊕ **Notification Levels:** Observers can listen to different events
- ⊕ **Group Observation:** Observers can listen to all objects of a class
- ⊕ **Dynamic Adaptation:** Subject does not have to implement an interface

Visitor

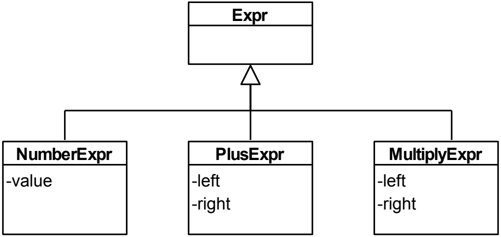
Pattern Description



- **Purpose:** Adding new operations to a family of classes
- **Mechanism:** Separate *visitor* class, back-and-forth interaction (*double dispatch*) between objects and visitor
- **Problem:** Complex object interaction (double dispatch)

Visitor

Example



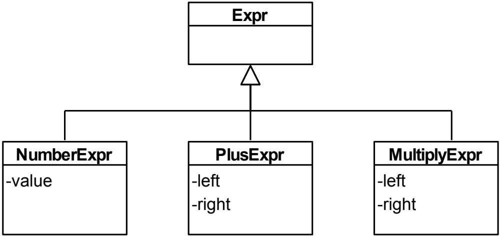
Visitor

Traditional Implementation: Example

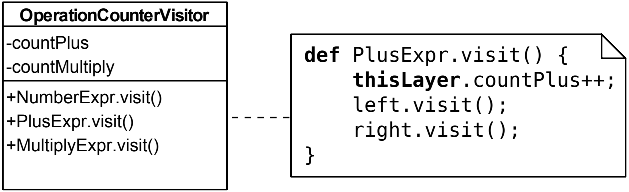
```
class PlusExpression extends Expression {
  def left, right;
  def accept(visitor) { visitor.visitPlusExpr(this); }
}

class NumberExpression extends Expression {
  def value;
  def accept(visitor) { visitor.visitNumberExpr(this); }
}

class OperationCounterVisitor {
  def countPlus, countNumber;

  def visitPlusExpr(node) {
    this.countPlus++;
    node.left.accept(this); node.right.accept(this);
  }

  def visitNumberExpr(node) { this.countNumber++; }
}
```

Visitor

COP Implementation: Example



- A visitor is an object that provides partial methods for new operations
- Partial methods can call visitor methods on other objects directly



Visitor

COP Implementation: Example

```
def treeRoot = /* ...  */
def visitor = new OperationCounterVisitor();

// Activate visitor in a block scope
with (visitor) {
  def result = treeRoot.visit();
}
```

Visitor

COP Implementation: Consequences

- ⊖ **Composability:** Potential name clashes between simulataneouly activated visitors (but visitors can use different method names)
- ⊕ **Simple Object Interaction:** No double dispatch necessary
- ⊕ **Dynamic Adaptation:** Classes do not have to provide accept methods

Overview

Summary

- **Classes as Layers:** Partial methods are members of classes and classes are instantiable
- COP Implementation of **Design Patterns**
  - *Decorator:* layer instance with partial methods for decorated methods
  - *Observer:* layer instance with partial methods for methods triggering state changes
  - *Visitor:* layer instance with partial methods for new operations
- Design patterns are not mere refactorings and have different semantics
- **Future work:** Implementation, analysis of other GoF design patterns, language features (e.g., partial method visibility), performance optimizations

References

[1] M. Appeltauer, R. Hirschfeld, J. Lincke. Declarative Layer Composition with the JCop Programming Language. Journal of Object Technology, Vol. 12, 2013

[2] M. Springer, H. Masuhara, R. Hirschfeld. Hierarchical Layer-based Class Extensions in Squeak/Smalltalk. Modularity Companion 2016.

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

[4] J. Lincke, M. Appeltauer, B. Steinert, R. Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. Science of Computer Programming, 2011.

Appendix

Method Lookup

- **Design Patterns:** How can we write an abstract visitor?
- **Language Semantics:** What happens if we override a partial method?
- **3 Dimensions:** Receiver class inheritance, layer inheritance, layer composition

Visitor

Overwriting Partial Methods: Layer Subclassing

```
class Evaluator
  def PlusNode.visit() {
    return left.visit() + right.visit();
  }
}

class ModEvaluator extends Evaluator {
  def modulo;

  ModEvaluator(def modulo) {
    this.modulo = modulo;
  }

  @override
  def PlusNode.visit() {
    return super.visit() % thisLayer.modulo;
  }
}
```

Visitor

Overwriting Partial Methods: Polymorphic Overriding

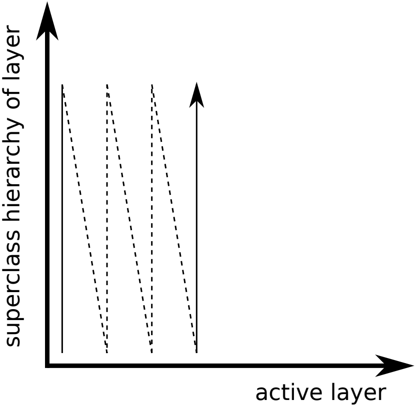
```
class SomeVisitor
  def Node.visit() {
    return /* ...  */
  }

  @override
  def PlusNode.visit() {
    return super.visit() + /* ...  */;
  }
}
```

Visitor

Overwriting Partial Methods: Layer Composition

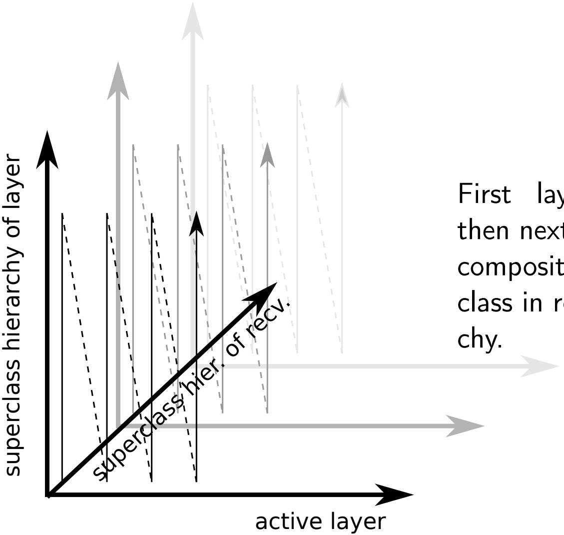
```
class SomeVisitor
  def Node.visit() {
    return /* ...  */
  }
}

class AnotherVisitor
  def Node.visit() {
    return super.visit() + /* ...  */;
  }
}

with (new Visitor()) {
  with (new AnotherVisitor()) {
    node.visit();
  }
}
```

Method Lookup



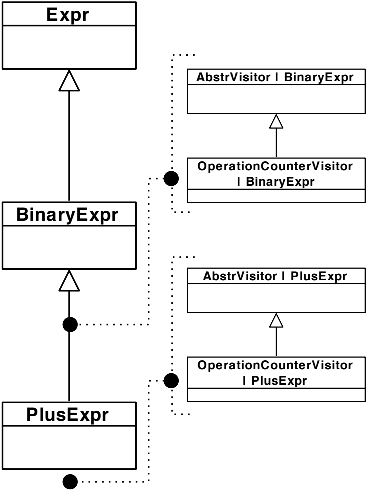
First layer hierarchy, then next class in layer composition.

Method Lookup



First layer hierarchy, then next class in layer composition, then next class in receiver hierarchy.

Method Lookup



$$LayerHierarchy(L, C) = \sum_{i=0}^{\#L} \langle super^i(L)[C] \rangle$$

$$ClassLayers(C) = \left(\sum_{i=1}^{|S|} LayerHierarchy(S[i], C) \right) + \langle C \rangle$$

$$Effective(C) = \sum_{i=0}^{\#C} ClassLayers(super^i(C))$$