Аспектно-ориентированное программирование

Денис С. Мигинский

Концепции АОП

Как парадигма программирования:

- Расщепление классов на несколько независимых частей
- Расщепление методов
- Изменение поведения в зависимости от контекста вызова («контекстный полиморфизм»)

Как методология:

- Единицей модульности является аспект, при этом способ выделения аспекта не фиксируется: вопрос «что первично, поведение или состояние?», решается для каждой отдельной задачи или подзадачи, а не всей парадигмы
- Классы, функции могут быть составными

Задача: расширение поведения

Требуется расширить поведение классов так чтобы:

- Перед/после всех или некоторых методов выполнялись определенные действия (журналирование, авторизация и т.д.)
- Классы ничего не знали о таком расширении (замечание: на расширение функциональности также распространяется LSP)
- Соблюдался принципе DRY при определении одинакового поведения для нескольких функций/классов

Варианты решения

1. Создание подкласса.

Плюсы: реализуемо почти в любом языке

Минусы: проблемы при инстанцировании, дублирование кода

2. Создание proxy-класса

Плюсы: реализуемо почти в любом языке, можно определить proxy на целую иерархию классов

Минусы: проблемы при инстанцировании, до конца не решает проблемы дублирования кода (решается MOP, если он поддерживается)

3. Использование вспомогательных методов

Плюсы: нет проблем при инстанцировании

Минусы: мало языков с поддержкой методов, до конца не решает проблемы дублирования кода

4. Кодогенерация, инструментирование байт-кода

Плюсы: полностью решает задачу

Минусы: без специализированных фреймворков реализация очень сложна

Взаимодействие с Clojure с Java

Java-вызовы из Clojure:

Прямой доступ к инструментарию Java: инстанцирование классов, вызов методов, обращение к полям и т.д., синтаксический сахар для инстанцирования классов, реализующих, фактически, функции высшего порядка (Thread, Future и т.д.)

Обращение из Java к произвольному динамическому языку:
Java Scripting API

Обращение из Java к Clojure:

Компиляция классов/интерфейсов (в том числе динамическая)

Генерация Java-класса из пространства имен Clojure

```
(ns ru.nsu.fit.dt.ClojureClass
  (:gen-class    ;explicitly generate class-file
    :state state ;accessor for state
    :init init   ;init function
    :constructors {[String] []}
    :main false
    :prefix impl-
    :methods [[printHello [] void]
              ^:static [staticPrintHello [] void]]))

(defn impl-init [s]
  [[] (atom s)])

(defn impl-printHello [this]
  (println "ClojureClass call:" @(.state this)))

(defn impl-staticPrintHello []
  (println "ClojureClass static call"))
```

Вызов из Java

```
/*
 * Предварительно следует убедиться
 * что Clojure-классы сгенерированы.
 * Для Eclipse: установить зависимость от Clojure-проекта,
 * в build-path добавить его директорию classes
 */

public class MainClass {
  public static void main (String[] args){
    ClojureClass.staticPrintHello();
    new ClojureClass ("java call").printHello();
  }
}

>>
ClojureClass static call
ClojureClass call: java call
```

Решение задачи: генерация proxy-класса

```
(ns ru.nsu.fit.dt.ProxyGenerator
  (:gen-class
    :main false
    :prefix impl-
    :methods [^:static [wrap [Object Class] Object]]))


(defn impl-wrap [obj iface]
  (println "Wrap target iface:"(.getName iface))
  ;;эта функция будет генерировать класс
  (gen-proxy-class iface)
  ;;а эта – создавать композицию из proxy и объекта
  (wrap-obj iface obj))
```

Код для генерации класса

```
;;;генерируем класс
(defrecord SomeInterface_proxy
  ;;;атрибуты
  [obj]
  ;;;реализуемый интерфейс
  SomeInterface
  ;;;метод реализуемого интерфейса
  (someMethod [msg]
    (println (.getName (class obj)) ": someMethod called")
    (. obj someMethod msg)))

(->SomeInterface_proxy obj)


;;;Вышеприведенный код должен генерироваться автоматически
;;;по переданному дескриптору интерфейса
```

Вспомогательные функции

```
;;;вычисляем имя (строку) SomeInterface_proxy
(defn- proxy-nm [iface]
  (.concat (.getSimpleName iface) "_proxy"))

;;; вычисляем символ SomeInterface_proxy
(defn- proxy-sym [iface]
  (symbol (proxy-nm iface)))

;;; вычисляем символ конструктора ->SomeInterface_proxy
;;; после eval получаем функцию-конструктор, как объект
(defn- proxy-cons [iface]
  (eval (symbol (.concat "->" (proxy-nm iface)))))

;;; конструируем proxy
(defn wrap-obj [iface obj]
  ((proxy-cons iface) obj))
```

Кодогенерация

```
(defn gen-proxy-class [iface]
  (eval (concat
          (list 'defrecord
                (proxy-sym iface)
                ['obj]
                (symbol (.getName iface)))
          ;;вычисляем методы интроспекцией
          (for [m (.getMethods iface)]
            (let [m-name (.getName m)
                  m-sym (symbol m-name)]
              ;;упрощение жизни: считаем, что все методы
              ;;от одного аргумента
              (list m-sym '[this msg]
                    ;;расширение функциональности
                    (list println '(.getName (class obj))
                          ":" m-name "called")
                    ;;вызов исходного метода
                    (list '. 'obj m-sym 'msg)))))))
```

Анализ решения

Плюсы:

- Относительно простое решение (в сравнении с прямой кодогенерацией или инструментированием байт-кода)
- Обеспечивается DRY
- Не специфично для Clojure: может быть воспроизведено в JRuby, Groovy для JVM, во многих динамических языках для CLR

Минусы:

- Теряем в производительности при вызове
- Требуется явное «оборачивание» (проблема устраняется при использовании DI)

Понятие компонентно-ориентированного программирования

Основа:

Объектно-ориентированное программирование

Мотивация:

Связь моделей через базовые классы (имеющие реализацию) увеличивает хрупкость системы

Дополнительные ограничения (к ООП):

Наследование в общем виде запрещено

Разрешается только имплементация классом (компонентой) интерфейса(-ов)

Объектная модель Clojure: протоколы, типы, записи

```
;;;генерация класса во время компиляции
;;;аналогично :gen-class
(gen-class …)

;;;генерация интерфейса во время компиляции
(gen-interface …)

;;;декларация протокола/генерация интерфейса
;;;во время исполнения
(defprotocol MyProto
  (method1 [this])
  (method2 [this] [this y]))

;;;генерация классов (как реализаций протоколов/интерфейсов)
;;;во время исполнения
(deftype …)
(defrecord …)
```

Аспектно-ориентированное программирование: предпосылки

Методологические проблемы:

Разделение ответственностей 2-го класса (**cross-cutting concerns**)

Технологический прототип:

Common Lisp Object System

Meta-Object Protocol

Первая «каноническая» реализация:

Язык AspectJ, автор Грегор Кичалес, Xerox PARC

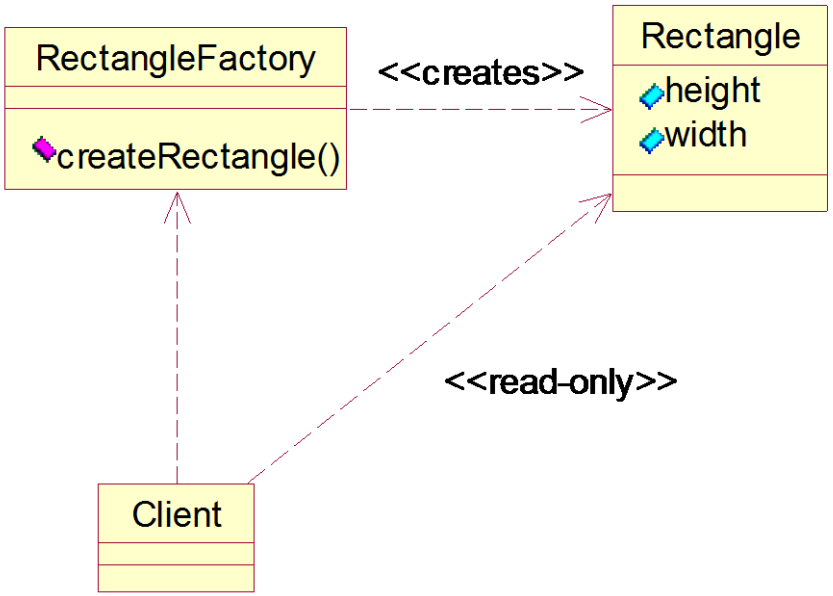
Инструментарий для разработки:

AspectJ Development Tool for Eclipse (AJDT)

Основные понятия AspectJ

- Join point (точка выполнения) – любая идентифицируемая точка программы (во время компиляции и или выполнения)
- Pointcut (срез) – набор (класс) точек выполнения программы, шаблон которому удовлетворяет этот набор точек
- Advice – изменение функциональности, применяемое к точке выполнения
- Inter-type declaration – дополнительное внешнее свойство уже существующего класса
- Aspect (аспект) – организационная сущность для всего вышеперечисленного

Задача: внешняя проверка контракта



Контракт:

- height, width >= 0
- Rectangle инстанцируется в RecatngleFactory
- Модификация Rectangle запрещена в Client

Задача:

Формально проверять контракт (AOT или JIT) без вмешательства в код

Решение на AspectJ

```
public aspect RectangleConstraints {

  pointcut rcSetter (double newval) :
    set (double Rectangle.*) && args (newval);

  void around (double newval) : rcSetter (newval){
    if (newval >= 0) proceed (newval);
    else proceed (0);
  }

  pointcut wrongInstance () :
    call (Rectangle.new(..)) && !within (RectangleFactory);

  declare warning: wrongInstance ():
    "Incorrect instantiation context for Rectangle";

  declare warning: set (double Rectangle.*) &&
    !within (RectangleFactory):
    "Incorrect modification context for Rectangle";
}
```

Задача: примеси в Java

```
package ru.nsu.fit.dt;

public interface IFoo {
  //хотим реализацию для этих методов
  public void printHello(String msg);
  public String doubleMsg(String msg);
}

//хотим включить IFoo как примесь в эти классы
public class Bar1{}

public class Bar2{}

public class Bar3{}
```

Решение

```
package ru.nsu.fit.dt.weave;
import ru.nsu.fit.dt.IFoo;

public aspect FooInjector {

  public void IFoo.printHello (String msg){
    System.out.println ("IFoo impl (" +
                this.getClass().getName() +
                "): " + msg);
  }

  public String IFoo.doubleMsg (String msg){
    return msg+msg;
  }

  declare parents: ru.nsu.fit.dt.Bar* implements IFoo;
}
```

Расширение иерархии

```
package ru.nsu.fit.dt;

public class Composite
  //фактически, включаем примесь, методы реализовать не
  //обязательно
  implements IFoo
{
  private IFoo part;

  public Composite (IFoo part){
    this.part = part;
  }

  public void printHello (String msg){
    System.out.println ("InnerComposite printHello:");
    part.printHello(msg);
  }
}
```

Решение задачи расширения поведения

```
public aspect CompositeInterceptor {

  pointcut interceptPrint (Object obj) :
    //перехватываемый вызов
    call (* *.println(..)) &&
    //условие на состояние стека вызовов
    cflow(execution(* ru.nsu.fit.dt.Composite.* (..))) &&
    //без этого получим бесконечную рекурсию
    !within(CompositeInterceptor) &&
    //связывание параметров
    this(obj);

  before (Object obj): interceptPrint (obj){
    System.out.println("INTERCEPT: println called in " +
                       obj.getClass().getName());
  }
}
```

Обзор AspectJ: основные виды pointcut

Аксессоры:

- get – обращение к полю
- set – присваивание полю

Вызов:

- call – точка вызова метода
- execute – точка входа в метод

Связывание аргументов:

- this – объект, из которого произошел вызов
- target – объект, к которому применен метод
- args – аргументы вызова

Контекст вызова:

- within – вызов из любого метода конкретного класса
- withincode – вызов из конкретного метода
- cflow, cflowbelow – состояние стека

Произвольное условие на аргументы:

- if

Виды advice

```
before (IFoo obj): somePointcut (obj){}

after (IFoo obj): somePointcut (obj){}

after (IFoo obj) returning (String res): somePointcut (obj){}

after (IFoo obj) throwing (Exception ex): somePointcut (obj){}

String around (IFoo obj) : somePointcut (obj){
  //…
  Object[] args = thisJoinPoint.getArgs();
  proceed(obj);
  //…
  return …
}
```

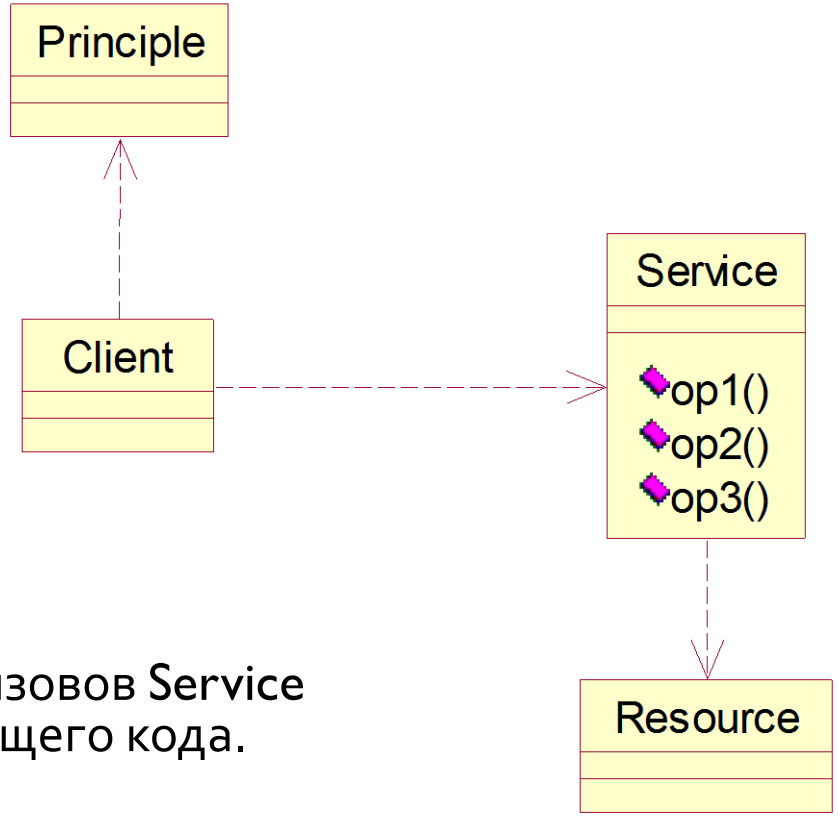
Основные декларации

```
String SomeClass.someMethod(String param){
  //…
}

declare parents: SomeClass extends BaseClass
                            implements SomeInterface;

declare warning: someStaticPointcut() "message";

declare error: someStaticPointcut() "message";
```

Другие возможности AspectJ

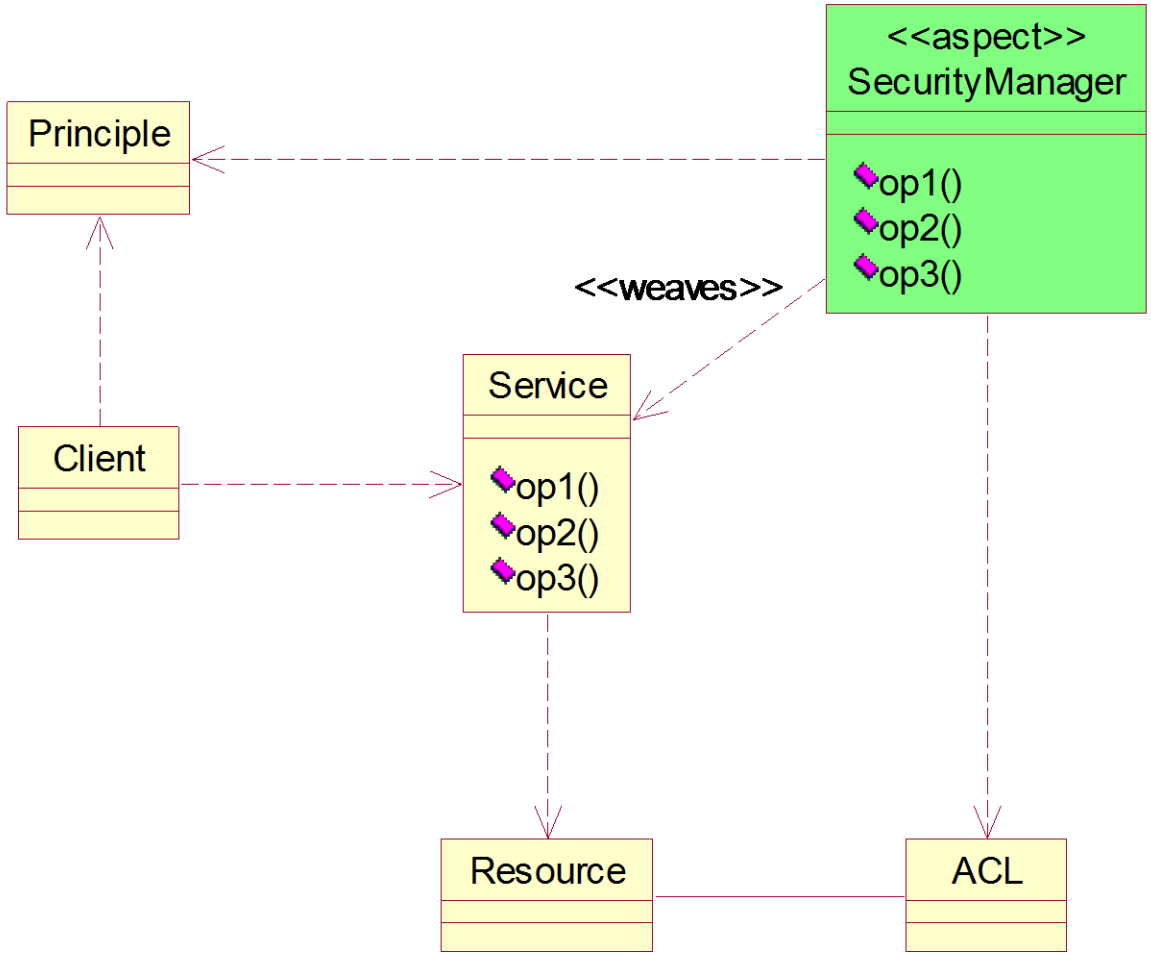
- Аспекты с состоянием, управление инстанцированием аспектов
- Управление порядком применения advises
- Абстрактные pointcuts
- Генерализация аспектов
- Использование Java-аннотаций в pointcuts
- Альтернативная форма языка – Java-аннотации (@Aspect и т.д.). Эти аннотации могут быть имплементированы в других фреймворках (Spring AOP и т.д.)

Пример проектирования: авторизация



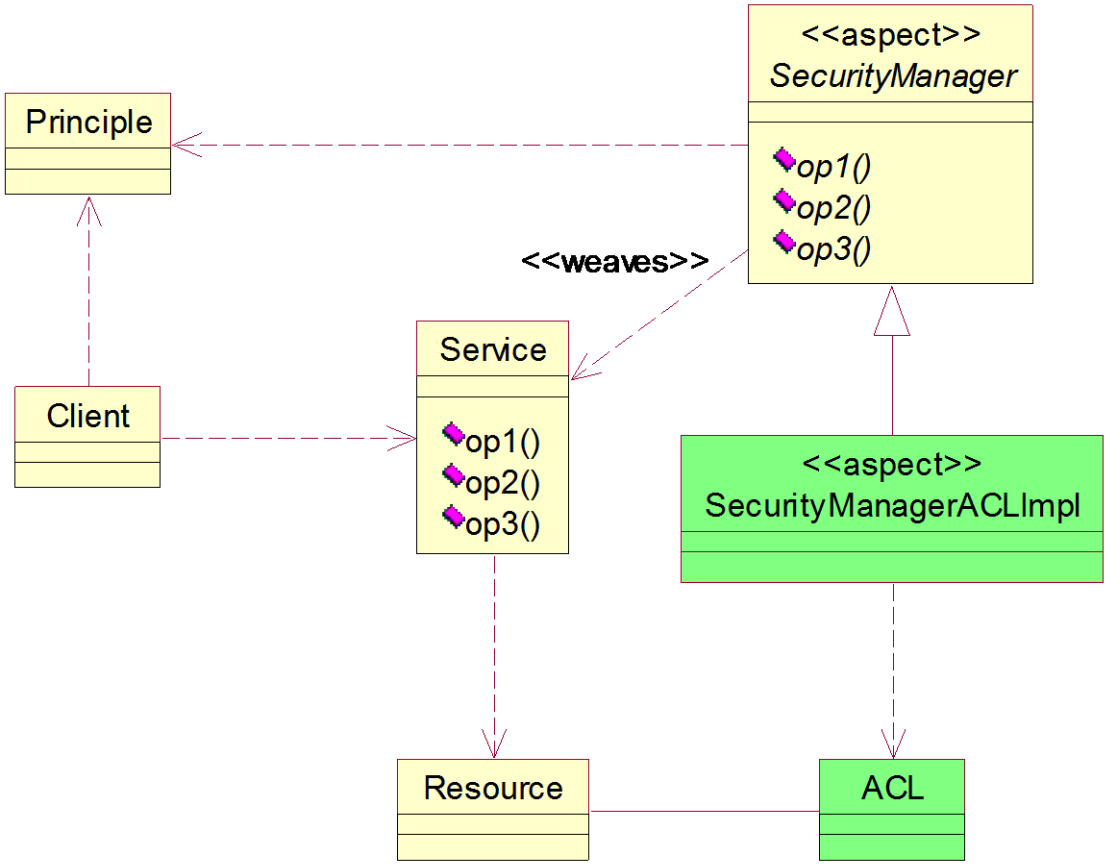
Требуется авторизация вызовов Service без изменения существующего кода.

Решение



Проблема: аспект определяет как poincuts, которые не зависят от модели безопасности, так и реализацию этой модели. Требуется разделить на два аспекта.

Решение с генерализацией аспектов



Применение абстрактных аспектов в проектировании

Базовый аспект без поведения

- Базовый (абстрактный) аспект определяет набор **pointcuts**, как точек расширения системы, но не определяет логику
- Производный аспект определяет логику (**advices, inter-type declaratiuons**)
- Таким образом, определяется точка расширения в аспектной форме. См. пример выше.

Базовый аспект без привязки к коду

- Базовый аспект определяет логику в привязке к абстрактным **pointcuts**
- Производный аспект определяет все необходимые **pointcuts**, привязывая таким образом логику к конкретному коду.
- Примеры: универсальные наблюдатели (Observer/Listener), механизмы Undo/Redo и т.д.