Разделение функциональности: наследование, модули, примеси

Андрей Васильев

2018

Принцип DRY

Do not Repeat Yourself - при разработке программного обеспечения необходимо устранять ненужное дублирование

Если логика изменится, то нужно исправить только в 1 месте

Примеры задач с общей логикой

- Разработка приложения, работающего с поставкой товаров. У каждого типа доставки есть общая логика, например, вычисление веса и габаритов посылки
- Разработка приложения, вычисляющего налоговые отчисления для заказов, товаров, услуг

В каждом из этих случаев следует избегать дублирования

Классы

Классы описывают общую логику действий для множества различных объектов: методы класса доступны всем объектам

- Каждый объект ответственен за своё состояние
- Каждый объект имеет ссылку на класс, к которому он относится
- Методы хранятся в классе и не дублируются в объектах
- Классы являются глобальными объектами

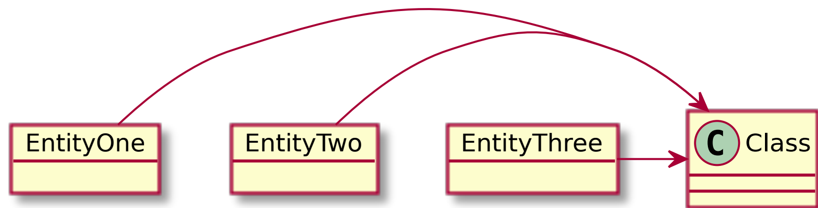

Рис.: Множество объектов одного класса

Базовая логика обработки сообщений

Вызов метода в Ruby - передача сообщения, состоящего из

- Имени метода, который вызывается у объекта
- Переданных аргументов

Когда объект получает сообщение, то оно передаётся классу данного объекта для обработки

- Если метод был найден, то он вызывается на объекте
- Если метод не был найден, то генерируется исключение

Поиск метода происходит только по имени, из аргументов учитывается только лишь их количество

Наследование и сообщения

Механизм наследования позволяет создавать новые версии класса, являющиеся его уточнением или специализацией

- Новые классы называются наследниками (подкласс)
- Исходный класс называется родителями (суперкласс)
- При наследовании подкласс наследует все возможности суперкласса - все методы доступны
- У класса может быть только 1 родитель

Пример наследования

Синтаксис наследования

```
class Parent
  ...
end
class Child < Parent
end
```

- Для описания наследования используется символ <
- Символ говорит о том, что класс слева является специализированной версией класса справа
- Класс `Child` является подклассом `Parent`
- Все методы суперкласса доступны в подклассе
- `self` содержит ссылку на объект, на котором вызван метод

Пример кода

```
class Parent
  def say_hello
    puts "Hello from #{self}"
  end
end
parent = Parent.new
puts parent.say_hello

class Child < Parent
end
child = Child.new
puts child.say_hello
```

Дерево наследования

- Для исследования дерева наследования можно воспользоваться методом `superclass`
- Для подкласса данный метод возвратит ссылку на родительский класс
- Если нет явного суперкласса, тогда класс унаследован от класса `Object`
- Суперклассом для `Object` является `BasicObject`, у которого нет суперкласса

```
puts "Child superclass: #{Child.superclass}"
puts "Parent.superclass: #{Parent.superclass}"
puts "Object.superclass: #{Object.superclass}"
puts "BasicObject.superclass "\
     "#{BasicObject.superclass.inspect}"
```

Последовательность вызова метода

Напомним, что общение между объектами в Ruby напоминает отправку сообщения: в сообщении указывается название метода и его аргументы.

Когда объект получает сообщение, то он начинает поиск метода, которому можно передать данное сообщение:

- Если метод есть у класса объекта, то вызывать его
- Если метод есть у суперкласса, то вызвать его
- Если метод есть у суперкласса суперкласса...

Первый найденный метод в данной цепочке будет вызван и ему будут переданы все переданные аргументы

Перекрытие методов в подклассах

- Вы можете в подклассе создать новый метод с именем метода из суперкласса, тогда он будет вызван раньше метода из суперкласса и перекроет последний
- Если семантика метода будет сохранена, тогда вызывающий класс не отличит оригинальный метод от подмены
  - Количество аргументов и их назначение
  - Возвращаемое значение и использование блока
- Для вызова метода из суперкласса используйте специальный метод `super`, который является альтернативным названием родительского метода

Метод to_s класса Object

- Метод преобразует объект к строке, используется стандартными методами вывода на потоки
- Рекомендуется реализовать свой метод `to_s`, преобразующий текущий объект к строке

```ruby
class Person
  def initialize(name)
    @name = name
  end
end
person = Person.new("Michael")
puts person # <Person:0x007fc812839550>

class Person
  def initialize(name)
    @name = name
  end
  def to_s
    "Person named #{@name}"
  end
end
person = Person.new("Michael")
puts person # Person named Michael
```

Проблема роста приложений

При написании сложных приложений разработчики зачастую сталкиваются с проблемой роста:

- Необходимо придумывать уникальные имена классов, не используемые в других частях приложения и библиотеках
- Необходимо логически объединять методы, не принадлежащие ни к одной реальной сущностия
- Необходимо логически объединять классы, предназначенные для решения одной большой задаче

В `Java` *проблема решается обязательным выделением пакетов, а в последней версии и модулей со встроенными версиями.*

Подходы к решению проблемы

- Правильное распределение элементов по файлам
- Использование классов для описания сущностей
- Объединение классов и функций в модули

Модули в языке Ruby

Модуль - замкнутое именованное пространство

```
module Trig
  PI = 3.141592654
  def Trig.sin(x) # Определяем метод модуля
    # ..
  end
  def Trig.cos(x)
    # ..
  end
end

puts Trig::PI # Доступ к константе
puts Trig.sin(Trig::PI/4) # Вызов метода
```

- Для доступа к константам, определённым внутри метода используются два двоеточия `::`
- Для вызова метода модуля используйте точку

```
module Moral
  VERY_BAD = 0
  BAD = 1
  def Moral.sin(badness)
    # ...
  end
end
y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

Определение методов в модуле

Определение методов модулей похоже на определение методов классов (не методов экземпляров классов). Можно даже использовать синтаксис с ключевым словом `self`

```
module ModuleWithBigName
  def ModuleWithBigName.greet(name)
    puts "Hello, #{name.upcase}"
  end
  def self.info
    'I am working'
  end
end
ModuleWithBigName.greet('mariya')
puts ModuleWithBigName.info
```

Оба метода в примере являются методами одного модуля

Определение констант в модуле

К константам, которые можно определить в модуле, относятся также модули и классы

```
module External
  class Internal
    ...
  end
end
object = External::Internal.new
```

Ввиду того, что модули тоже являются константами, то их содержимое можно определять в нескольких файлах, эффективно создавая замкнутые пространства имён

Проблемы множественного наследования

Предположим, что мы разрабатываем библиотеку для создания графического пользовательского интерфейса и мы определили следующие сущности: базовую и двух наследников

- Базовый графический элемент, определяющий размеры
- Наследник, позволяющий размещать элементы в колонку
- Наследник, позволяющий прокручивать своё содержимое

Если мы захотим создать новый элемент, который реализует прокручиваемый список элементов, то нам будет необходимо унаследоваться от последних двух классов. Вопросы:

- Сколько раз будут создаваться и настраиваться данные базового класса?
- Как будут вести себя переменные, которые используются одновременно в двух классах-наследниках?
- Что произойдёт с методами, имеющими одинаковыме названия, но разную реализацию?

Mixin (Примеси, Миксины, Агрегация)

Модули позволяют решить вопросы множественного наследования, реализуя возможность агрегирования: модули можно примешивать к определениям классов

```
module Debug
  def who_am_i? # Экземпляр метода
    "#{self.class.name} (id: #{self.object_id})"
  end
end
class Test
  include Debug # Примешивание модуля Debug
end
Test.new.who_am_i?
```

- Методы примеси определены как методы экземпляра
- Данные методы становятся методами экземпляра класса
- Модуль примешивается с помощью метода `include`

Особенности примеси модулей

- `include` не подключает файл, а только ссылается на описанный ранее модуль. Если примесь определена в отдельном файле, то его необходимо подключить
- `include` не копирует методы из модуля в класс, а только лишь добавляет ссылку в класс
  - Если несколько классов примешивают в себя один модуль, тогда они все ссылаются на один и тот же модуль
  - Если вы изменяете такой модуль, то «автоматически» меняются и все классы, что примешивают данный модуль
- `include` подключает только лишь методы экземпляра, для подключения методов модуля в методы класса необходимо использовать метод `extend`

Пример использования extend

```
module Mixin
  def self.class_method
    puts 'In the class method'
  end
  def instance_method
    puts 'In the instance method'
  end
end
class Test
   extend Mixin
end
Test.class_method
test = Test.new
test.instance_method # Не сработает
```

Другие схемы наследования

Множественное наследование (C++)

- У класса может быть множество родительских классов
- Каждый из родительских классов может полноценно реализовывать свою логику

Один класс-родитель (Java < 8, C#)

- У класса может быть только один родительский класс
- Класс может реализовывать множество интерфейсов

В Java 8 интерфейсы могут содержать код, который может выполняться. Это открывает возможности по использованию их в качестве примесей, однако такой практики пока ещё не сложилось.

Использование примеси Comparable

Язык Ruby включает в себя ряд полезных примесей, которые могут пригодится при создании собственных объектов

Примесь `Comparable` опирается на то, что в классе будет реализован метод сравнения <=>. Данный метод берёт ссылку на другой объект, сравнивает и возвращает

- Положительное число, если текущий объект «больше»
- Ноль, если текущий объект «равен»
- Отрицательное число, если текущий объект «меньше»

Большинство встроенных типов данных реализуют данный метод.

Используя этот метод примесь добавляет в ваш класс следующие методы: <, <=, ==, >, >=, `between`, `clamp`.

Пример использования примеси Comparable

```ruby
class SizeMatters
  include Comparable
  attr :str
  def <=>(other)
    str.size <=> other.str.size
  end
  def initialize(str)
    @str = str
  end
end

s1 = SizeMatters.new("Z")
s2 = SizeMatters.new("YY")
s3 = SizeMatters.new("XXX")
s1 < s2                          #=> true
s4.between?(s1, s3)              #=> false
```

Использование примеси Enumerable

Модуль `Enumerable` предоставляет большое число методов для работы с данными, которые можно перечислить.

Классу, использующему модуль Enumerable необходимо реализовать следующие методы:

- итератор `each` для обхода всех элементов коллекции
- объекты коллекции метод сравнения <=>

После примеси модуля становятся доступными все его методы:

- `map`
- `find`
- `reduce`
- `sort`
- ...

Проектирование примесей

Из рассмотренных выше примесей мы можем вынести следующие уроки:

- Примеси замечательно работают в случае, если на основании одного базового свойства вы можете построить целую систему
  - Для `Comparable` таким свойством явился метод `<=>`
  - Для `Enumerable` таким свойством явился метод `each`
- Примеси не изменяют внутренее состояние объекта напрямую, взаимодействуют только лишь с интерфейсом
- Примесь накладывает необходимые ей ограничения на класс, в который её необходимо примешать

Ошибки в проектировании примесей

Методы, примешанные к классу, могут взаимодействовать не только с методами, но также и с переменными экземпляра

```
module First
  def foo
    @abc = 10
  end
end
module Second
  def bar
    @abc = 'abc'
  end
end
```

Не следует разрабатывать модулей, которые оперируют состоянием переменных экземпляра напрямую

Какой метод будет вызван?

Когда у вас появляется возможность использовать примеси, то должен встать вопрос: как происходит поиск методов среди всех связных компонент? Порядок достаточно простой:

- Методы класса, к которому принадлежит объект
- Методы примешанных модулей
- Методы суперкласса
- Методы примешанных в суперкласс модулей
- ...

Вопрос для самоизучения

Зависит ли порядок поиска методов в примесях от порядка их включения в класс?

Использование примесей и наследования

Механизм наследования позволяет определить чёткую иерархию свойств объектов. Для любого класса-ребёнка должно выполняться правило: класс-ребёнок является классом-родителем. Т.е. в программе можно заменить класс-родитель на класс-ребёнок и приложение должно продолжить правильно функционировать.

В большинстве случаев реального мира мы имеем дело с предметами, обладающими множеством различных свойств и множеством контрагентов. То есть объект живёт в некоторой среде и комбинирует в себе качества множества различных компонент. Такое поведение обычно сложно представить в виде подмножества какого-то конкретного класса.

Простое правило, которое следует использовать при проектировании классов: «композиция лучше наследования»