Ruby-Erweiterungsmuster

Rails-Konferenz 2008
Tammo Freese, 10.06.2008

© 2008 Tammo Freese

Über mich

- Tammo Freese
- Technischer Fokus: Ruby/Rails, Objective-C/Cocoa
- Extreme Programmer
- Doktorand
- Freiberufler
- Zurzeit bei XING

© 2008 Tammo Freese

Überblick

- Erweiterungen in Ruby
  - Wiederkehrende Probleme
  - Allgemeine, wiederverwendbare Lösungen
- Erweiterungsmuster!
  - Acht oft verwendete Muster (nichts neues?)
  - Kein Anspruch auf Vollständigkeit

© 2008 Tammo Freese

Ruby-Erweiterungsmuster

- Problem:
  - Erweiterungsmuster sind interessant, aber zu grundlegend für diese Konferenz
- Lösung:
  - Stattdessen ein anderes (verwandtes) Thema :)
  - Folien zum ersten Thema stelle ich zur Verfügung

© 2008 Tammo Freese

Ruby-Erweiterungsregeln

Rails-Konferenz 2008
Tammo Freese, 10.06.2008

© 2008 Tammo Freese

Einleitung

- Ruby erweitern ist einfach, aber es wird schwieriger, wenn man dabei andere Erweiterer berücksichtigt.
- Drei Beispielprobleme für sieben Erweiterungsregeln:
  - String#camelize (neue Methode)
  - Array#sort! (erweitert mit Logging)
  - Array#find_by_... (über method_missing)

Sieben Ruby-Erweiterungsregeln

1. Care about others
2. Care about return values
3. Care about parameters
4. Care about exceptions
5. Extend, don't redefine
6. Care about reuse
7. Care about extensibility

© 2008 Tammo Freese

Beispiel 1: String#camelize

```
class String
  def camelize
    gsub(/(^|_)(.)/) { $2.upcase }
  end
end
```

Was ist das Problem?

- Problem:
  String wird überall verwendet. Was passiert, falls eine andere Erweiterung String#camelize anders definiert?
  - Vielleicht schwer nachvollziehbare Fehler
  - Vielleicht keine Fehler, aber falsches Verhalten
- Lösung:
  - Exception werfen, falls Methode schon definiert
  - Nur dann, wenn Konfliktwahrscheinlichkeit hoch!

© 2008 Tammo Freese

Lösung für Regel 1

```
class Class
  def all_instance_methods
    instance_methods | private_instance_methods
  end
  def assert_not_defined(method)
    return if !all_instance_methods.include?(method.to_s)
    raise "Name Collision for #{self}##{method}"
  end
end
```

© 2008 Tammo Freese

Lösung für Regel 1

```
class String
  assert_not_defined :camelize
  def camelize
    gsub(/(^|_)(.)/) { $2.upcase }
  end
end
```

© 2008 Tammo Freese

Regel 1 in freier Wildbahn

- Problem:
  - Array#sum (Rails mit anderen)
  - Object#metaclass (RSpec mit anderen)
- Lösung:
  - Builder::XChar in der Builder-Library

© 2008 Tammo Freese

Sieben Ruby-Erweiterungsregeln

1. Care about others
2. Care about return values
3. Care about parameters
4. Care about exceptions
5. Extend, don't redefine
6. Care about reuse
7. Care about extensibility

© 2008 Tammo Freese

Beispiel 2

```
class Array
  alias_method :sort_without_time_logging!, :sort!

  def sort!
    time_now = Time.now
    sort_without_time_logging!
    puts "sort!: #{Time.now - time_now} seconds."
  end
end
```

© 2008 Tammo Freese

Was ist das Problem?

- Problem:
  - Array#sort! gibt normalerweise das Array zurück.
  - Unsere Implementierung hat das geändert.
- Lösung:
  - Wir geben den Rückgabewert der ursprünglichen Implementierung zurück.

© 2008 Tammo Freese

Lösung für Regel 2

```
class Array
  alias_method :sort_without_time_logging!, :sort!

  def sort!
    time_now = Time.now
    result = sort_without_time_logging!
    puts "sort!: #{Time.now - time_now} seconds."
    result
  end
end
```

Regel 2 in freier Wildbahn

- Problem
  - tritt meist nur in der Entwicklungsphase auf
  - bleibt dort unentdeckt oder wird sofort behoben

© 2008 Tammo Freese

Sieben Ruby-Erweiterungsregeln

1. Care about others
2. Care about return values
3. Care about parameters
4. Care about exceptions
5. Extend, don't redefine
6. Care about reuse
7. Care about extensibility

© 2008 Tammo Freese

Was ist das Problem?

- Problem:
  - Array#sort! akzeptiert einen Block als Argument
  - Unsere Implementierung ignoriert Blöcke.
- Lösung:
  - Wir reichen den Blockparameter an die ursprüngliche Implementierung weiter.

Lösung für Regel 3

```
class Array
  alias_method :sort_without_time_logging!, :sort!

  def sort!(&block)
    time_now = Time.now
    result = sort_without_time_logging!(&block)
    puts "sort!: #{Time.now - time_now} seconds."
    result
  end
end
```

© 2008 Tammo Freese

Lösung für Regel 3

```
class Array
  alias_method :sort_without_time_logging!, :sort!

  def sort!(*args, &block)
    time_now = Time.now
    result = sort_without_time_logging!(*args, &block)
    puts "sort!: #{Time.now - time_now} seconds."
    result
  end
end
```

© 2008 Tammo Freese

Varianten der Parameterübergabe

- Kopie der ursprünglichen Parameter
  - Gute Dokumentation, arity bleibt erhalten
  - Verschluckt nicht-explizite Blöcke, kann andere Erweiterungen brechen
- (*args, &block)
  - Keine Dokumentation, verliert arity
- Beste Lösung: Kombination

© 2008 Tammo Freese

Regel 3 in freier Wildbahn

- Problem
  - tritt meist nur in der Entwicklungsphase auf
  - bleibt dort unentdeckt oder wird sofort behoben

© 2008 Tammo Freese

Sieben Ruby-Erweiterungsregeln

1. care about others
2. care about return values
3. care about parameters
4. care about exceptions
5. extend, don't redefine
6. care about reuse
7. care about extensibility

© 2008 Tammo Freese

Was ist das Problem?

- Problem:
  - Falls die ursprüngliche Implementierung von Array#sort! eine Exception wirft, loggt unsere Erweiterung die Zeit nicht.
- Lösung:
  - Falls eine Exception auftritt, die Zeit dennoch loggen.

© 2008 Tammo Freese

Lösung für Regel 4

```
class Array
  alias_method :sort_without_time_logging!, :sort!
  def sort!(*args, &block)
    time_now = Time.now
    result = sort_without_time_logging!(*args, &block)
  ensure
    puts "sort!: #{Time.now - time_now} seconds."
    result
  end
end
```

Regel 4 in freier Wildbahn

- Problem in RubyCocoa
  - Statt eine Exception der ursprünglichen Methode weiterzuwerfen, wurde eine neue von einem anderen Typ geworfen.
  - Konflikt mit RConditions
  - Patch wurde übernommen :)

© 2008 Tammo Freese

Sieben Ruby-Erweiterungsregeln

1. care about others
2. care about return values
3. care about parameters
4. care about exceptions
5. extend, don't redefine
6. care about reuse
7. care about extensibility

© 2008 Tammo Freese

Beispiel 3: Array#find_by_…

```
class Array
  def method_missing(id, *args)
    unless id.to_s =~ /^find_by_(.+)$/
      return super
    end
    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
end
```

Was ist das Problem?

- Problem:
  - Wir können nicht sicher sein, dass keine andere Erweiterung Array#method_missing erweitert.
  - Eine schon vorhandene Erweiterung würde durch unsere Implementierung überschrieben.
- Lösung:
  - Statt super verwenden wir alias_method.

Lösung für Regel 5

```
class Array
  alias_method :mm_without_find_by, :method_missing
  def method_missing(id, *args, &block)
    unless id.to_s =~ /^find_by_(.+)$/
      return super mm_without_find_by(id, *args, &block)
    end
    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
end
```

© 2008 Tammo Freese

Lösung für Regel 5?

- Problem:
  - Wenn zwei Erweiterungen auf find_by_… reagieren, welche soll den Vorzug bekommen?
  - Meine Meinung: First Come, First Serve
- Lösung:
  - Wir rufen zuerst die ursprüngliche Implementierung auf.

Lösung für Regel 5

```
class Array
  alias_method :mm_without_find_by, method_missing
  def method_missing(id, *args, &block)
    return mm_without_find_by(id, *args, &block)
  rescue NameError => e
    raise e unless id.to_s =~ /^find_by_(.+)$/
    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
end
```

© 2008 Tammo Freese

Regel 5 in freier Wildbahn

- Verletzungen sind einfach zu finden (z.B. method_missing-Implementierungen).
- Probleme mit den Verletzungen sind selten.

Sieben Ruby-Erweiterungsregeln

1. care about others
2. care about return values
3. care about parameters
4. care about exceptions
5. extend, don't redefine
6. care about reuse
7. care about extensibility

© 2008 Tammo Freese

Was ist das Problem?

- Problem:
  Unsere find_by_…-Erweiterung gilt nur für Array und kann nicht in anderen Kontexten wiederverwendet werden.
- Lösung:
  Wir implementieren unsere Erweiterung in einem Aspect Module (wie z.B. ActiveRecord::Validations).
- (Falls wir find_by_… nicht wiederverwenden möchten, dokumentiert das Modul die Erweiterung.)

Lösung für Regel 6

```
~~class Array~~ module FindBySupport
  ~~alias_method :mm_without_find_by, method_missing~~
  def ~~method_missing~~ mm_with_find_by(id, *args, &block)
    mm_without_find_by(id, *args, &block)
  rescue NameError => e
    raise e unless id.to_s =~ /^find_by_(.+)$/
    attr = $1
    find { |elem| elem.__send__(attr) == args.first}
  end
```

© 2008 Tammo Freese

Lösung für Regel 6

```
def self.included(base)
  base.class_eval do
    alias_method :mm_without_find_by, :method_missing
    alias_method :method_missing, :mm_with_find_by
  end
end
end
class Array
  include FindBySupport
end
```

Regel 6 in freier Wildbahn

- Regel 6 ist mehr eine Richtlinie, daher ist es schwer, von Verletzungen zu sprechen.

© 2008 Tammo Freese

Sieben Ruby-Erweiterungsregeln

1. care about others
2. care about return values
3. care about parameters
4. care about exceptions
5. extend, don't redefine
6. care about reuse
7. care about extensibility

© 2008 Tammo Freese

Was ist das Problem?

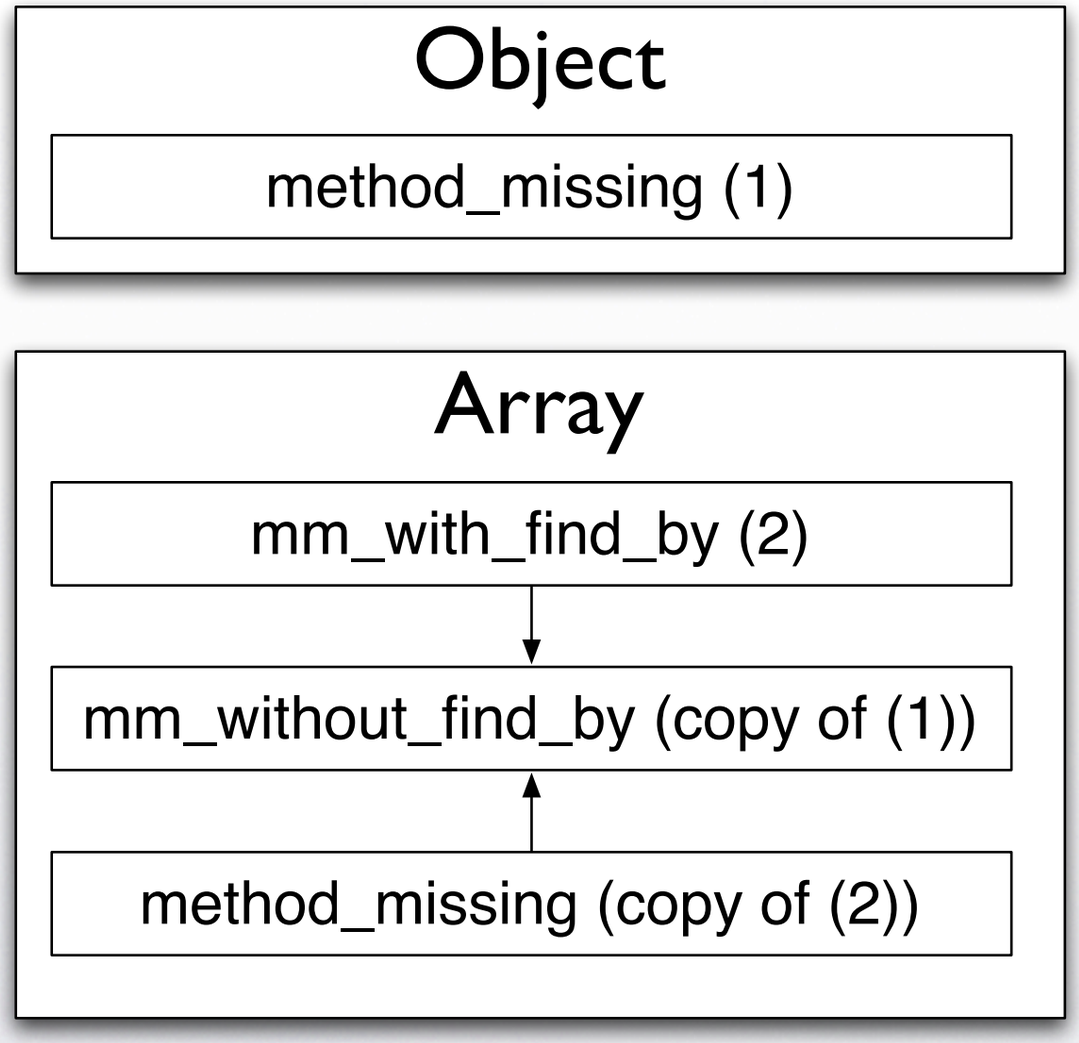
- Unsere Erweiterung sollte mit anderen Erweiterungen kombiniert arbeiten können.
- Unsere Erweiterung sollte selbst erweiterbar sein.
- Wir erweitern mit zwei alias_method-Aufrufen:
  alias_method :foo_without_bar, :foo
  alias_method :foo, :foo_with_bar
- Beide Aufrufe führen zu Problemen.

© 2008 Tammo Freese

Was ist das Problem?

- Problem:
  Falls eine andere Erweiterung Object#method_missing erweitert/redefiniert und sie nach unserer Erweiterung geladen wird, wird ihre method_missing-Implementierung nicht aufgerufen.
- Ursache:
  Unser erster alias_method-Aufruf kopiert die ursprüngliche Object#method_missing-Implementierung (Erklärung: nächste Folie)

Regel 7: Care About Extensibility



method from module

alias_method :mm_without_find_by,
:method_missing

alias_method method_missing,
:mm_with_find_by

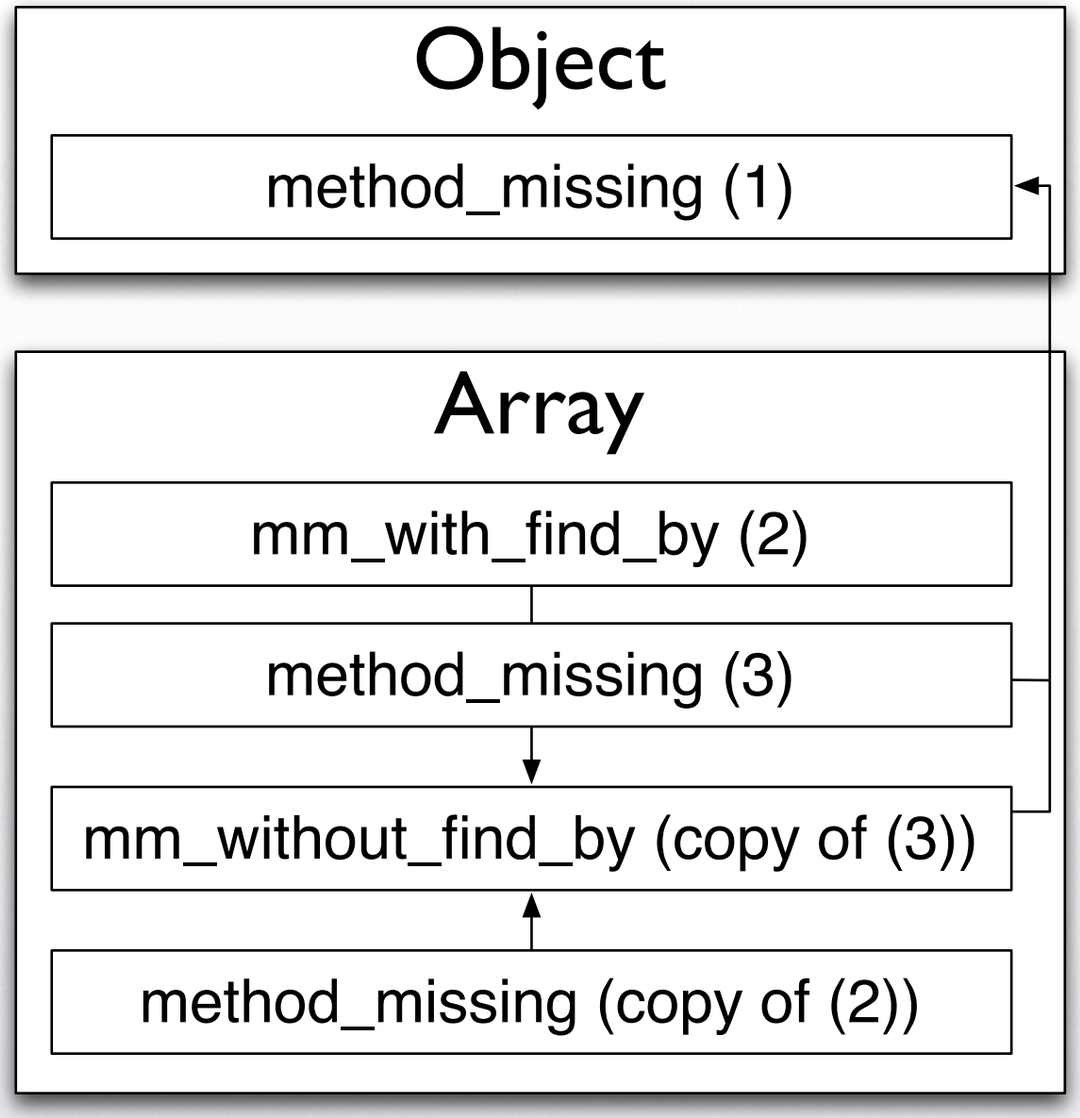
© 2008 Tammo Freese

Lösung für Regel 7

- Lösung:
  Vor dem ersten alias_method call implementieren wir Array#method_missing und rufen darin super auf:

```
def self.included(base)
  base.class_eval do
    def method_missing(id, *args); super; end
    alias_method :mm_without_find_by, :method_missing
    alias_method :method_missing, :mm_with_find_by
  end
end
```

Lösung für Regel 7



method from module

method_missing calling super

alias_method :mm_without_find_by, :method_missing

alias_method method_missing, :mm_with_find_by

Lösung für Regel 7?

- Ein Problem gelöst, ein anderes geschaffen:
  Falls eine andere Erweiterung
  Array#method_missing erweitert, und sie vor unserer Erweiterung geladen wird, würde unsere Implementierung sie überschreiben.
- Lösung:
  Wir definieren method_missing nur falls nötig.

Lösung für Regel 7

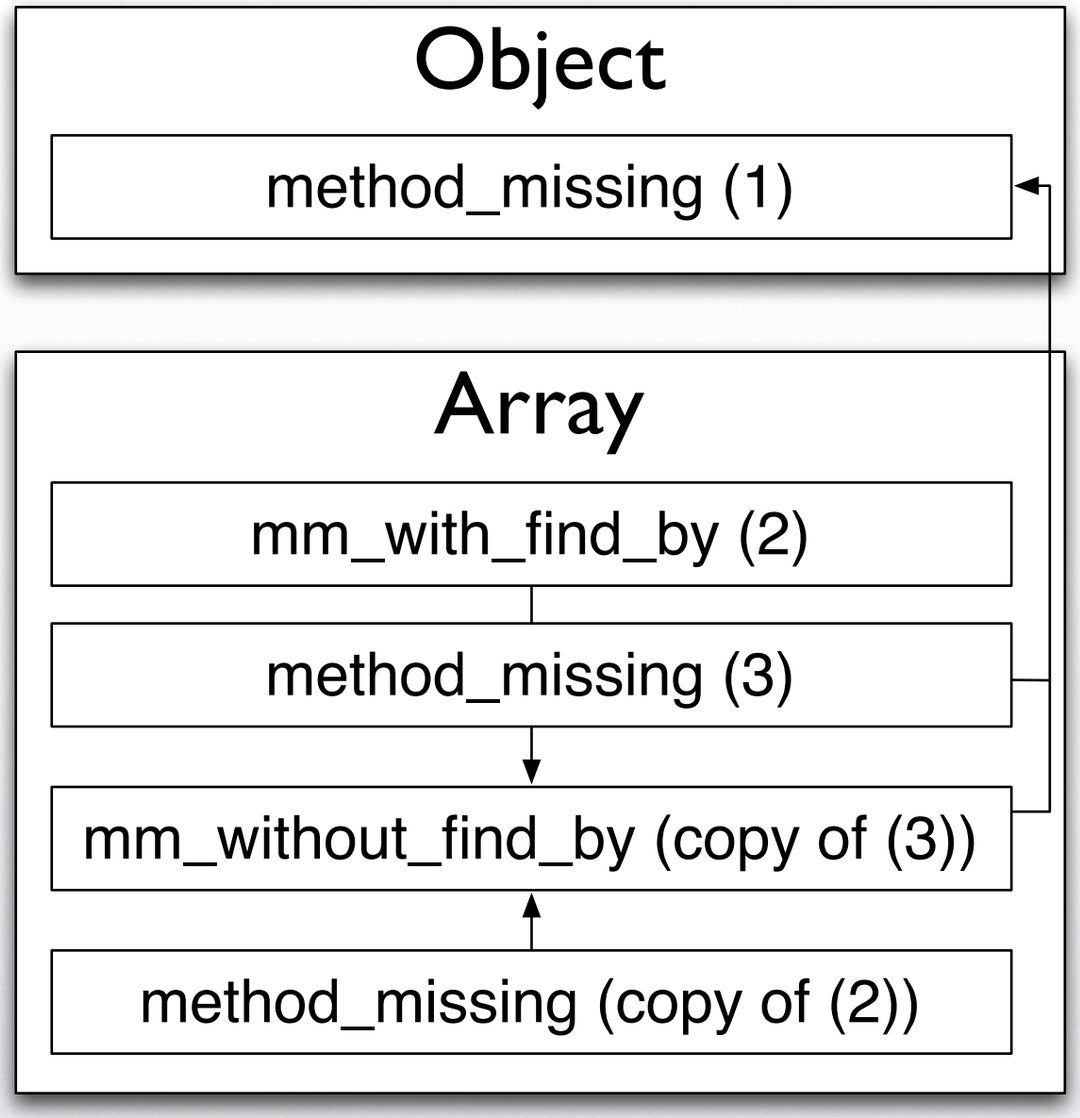
```
def self.included(base)
  base.class_eval do
    unless (instance_methods(false) +
        private_instance_methods(false)).include(
        'method_missing')
      def method_missing(id, *args); super; end
    end
    alias_method :mm_without_find_by, :method_missing
    alias_method :method_missing, :mm_with_find_by
  end; end
```

© 2008 Tammo Freese

Was ist das Problem?

- Letztes Problem:
  Erweiterungen von mm_with_find_by werden nicht aufgerufen.
- Ursache:
  Unser zweiter alias_method-Aufruf kopiert mm_with_find_by.

Regel 7: Care About Extensibility



method from module

method_missing calling super

alias_method :mm_without_find_by, :method_missing

alias_method method_missing, :mm_with_find_by

Lösung für Regel 7

- Lösung:
  Anstatt alias_method zu verwenden, redefinieren wir method_missing und rufen in der redefinierten Methode mm_with_find_by auf.

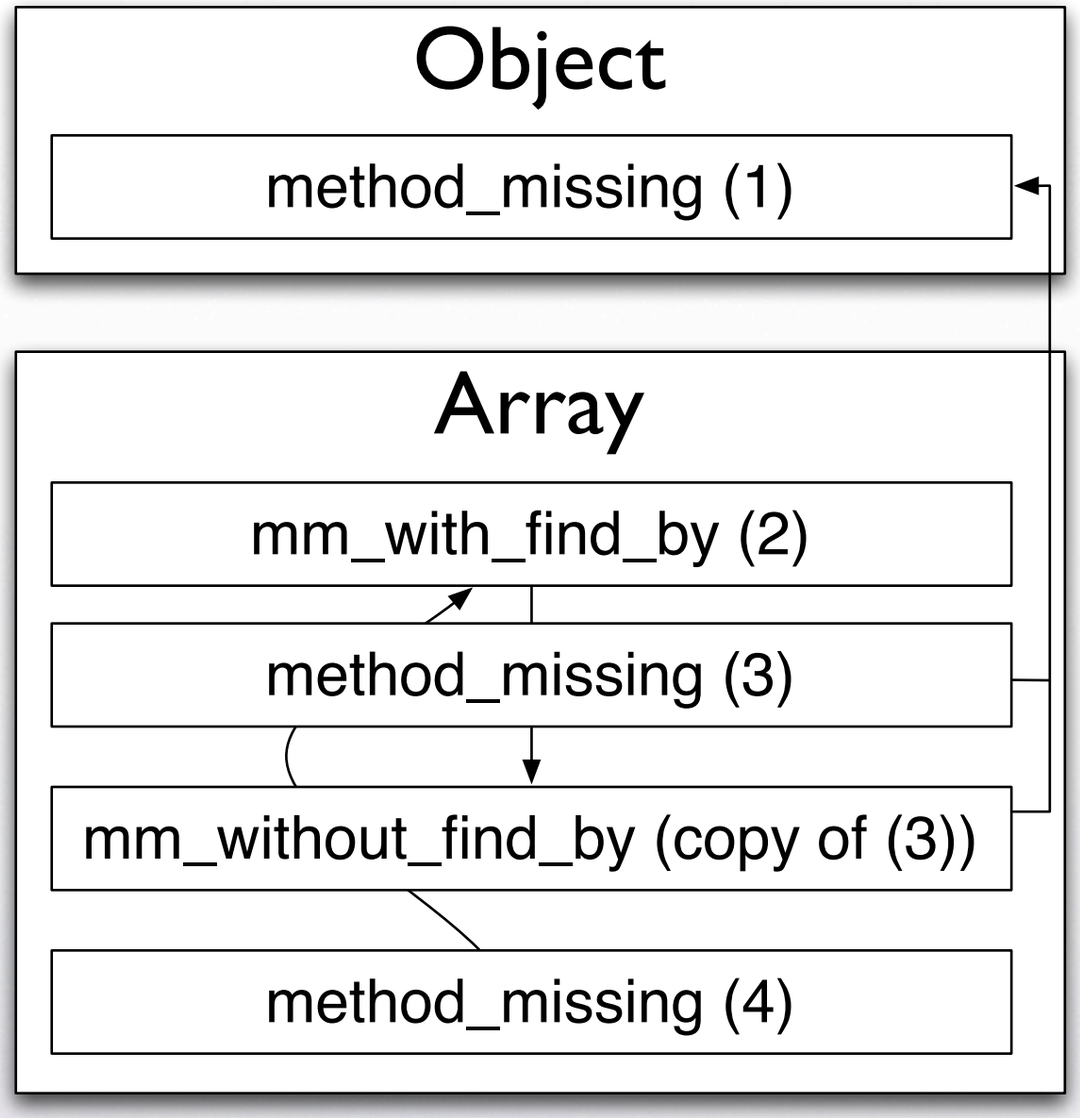
© 2008 Tammo Freese

Lösung für Regel 7

```
def self.included(base)
  base.class_eval do
    unless (instance_methods(false) … end
    alias_method :mm_without_find_by, :method_missing
    alias_method :method_missing, :mm_with_find_by
    def method_missing(id, *args, &block)
      mm_with_find_by(id, *args, &block)
    end
  end
end
```

© 2008 Tammo Freese

Lösung für Regel 7



method from module

method_missing calling super

alias_method :mm_without_find_by,
:method_missing

method_missing calling
mm_with_find_by

Regel 7 in freier Wildbahn

- Problem:
  - Builder's method_added Implementierung für BlankSlate (Module#method_added)
  - Ruby on Rails's alias_method_chain (…_with_…- Methoden können nicht erweitert werden)
- Fixes:
  - Builder: Patch hingeschickt, keine Antwort
  - Ruby on Rails: bitte +1 auf Issue #285

© 2008 Tammo Freese

Sieben Ruby-Erweiterungsregeln

1. care about others
2. care about return values
3. care about parameters
4. care about exceptions
5. extend, don't redefine
6. care about reuse
7. care about extensibility

© 2008 Tammo Freese

Die letzte Folie

- Falls Ihnen dieser Vortrag gefallen hat, bitte ich um eine Empfehlung auf workingwithrails.com .
- Fragen?
  - jetzt,
  - oder per E-Mail an tammo@tammofreese.de