Mapowanie obiektowe na przykładzie ***Ruby on Rails***

Autor:
Piotr Walkowski

Program wystąpienia

- Czym jest ORM / Active Records / MVC?
- Zalety i wady takiego podejscia
- Wprowadzenie do Ruby on Rails, pare słów o konwencjach
- Opis Active Records na przykładzie Ruby on Rails
- Tworzenie modeli / migracje
- Rodzaje połączeń - rodzaje
- Podstawowe operacje, transakcyjność
- Bardziej zaawansowane właściwości
- Sprawdzanie poprawności
- Rozbudowa modelu

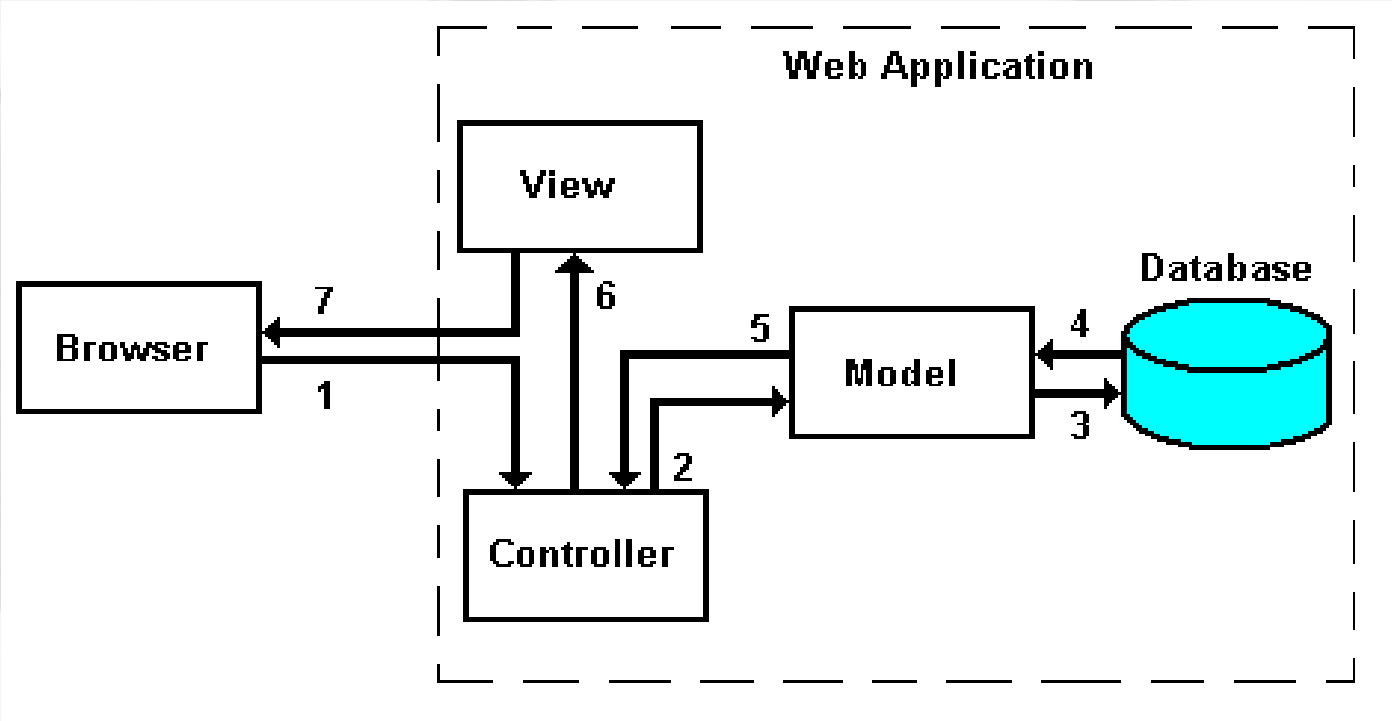
Object-Relational-Mapping

- **Mapowanie obiektowo-relacyjne** (*ang. Object-Relational Mapping ORM*) sposób odwzorowania obiektowej architektury systemu informatycznego na bazę danych (lub inny element systemu) o relacyjnym charakterze.
- Implementacja takiego odwzorowania stosowana jest m.in. w przypadku, gdy tworzony system oparty jest na podejściu obiektowym, a system bazy danych operuje na relacjach.

MVC

- **Model-View-Controller** (pol. Model-Widok-Kontroler) to architektoniczny wzorzec projektowy do organizowania struktury aplikacji
- Model-View-Controller zakłada podział na trzy główne warstwy:
  - Model - jest pewną reprezentacją problemu bądź logiki aplikacji.
  - Widok - opisuje, jak wyświetlić pewną część modelu w ramach interfejsu użytkownika.
  - Kontroler - przyjmuje dane wejściowe od użytkownika i reaguje na jego poczynania, zarządzając aktualizacje modelu czy odświeżenie widoków.

Ilustracja działania MVC



Kluczowe pojęcia: AR

- **Active Record** - *“In software engineering, the active record pattern is a design pattern found in software that stores its data in relational databases. It was named by Martin Fowler in his 2003 book Patterns of enterprise application architecture. The interface to such an object would include functions such as Insert, Update, and Delete, plus properties that correspond more or less directly to the columns in the underlying database table.”*

Przykłady zastosowania wzorca **Active Records**:

- *ADO.NET Framework (.NET)*
- ***Ruby on rails (ruby)***
- *Symfony (PHP)*
- *DBIx::Class (perl)*
- *Django (python)*

Ogólny przykład:

Przykład 1:
part = new Part()
part.name = "Sample part"
part.price = 123.45
part.save()

=> „INSERT INTO parts (name, price) VALUES ('Sample part', 123.45);”

Przykład 2:
b = Part.find_first("name", "gearbox")

*=>„SELECT * FROM parts WHERE name = 'gearbox' LIMIT 1;”*

Zalety takiego podejścia

- Bardziej zwarta całość (cała logika biznesowa, w tym relacje między tabelami, czy sprawdzanie poprawności jest w modelach)
- Prostrza i przyjazna składnia może zwiększyć wydajność
- Większe uniezależnienie od konkretnego DMBS
  (Np. można, w ogóle zrezygnować triggerów,
  badz procedur składniowych na rzecz „obserwatorów” klas modeli)
- Częściowo, poprawa bezpieczeństwa (np. uniknięcie SQL injection)

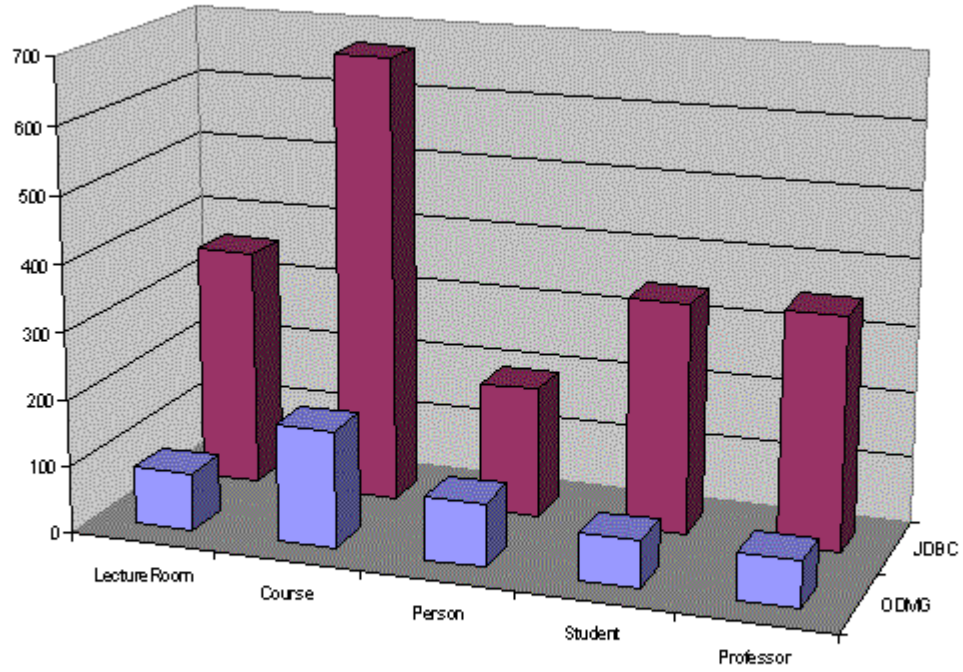
Zalety takiego podejścia:

- Większa “ekspresyjność”, w wyrażaniu (przy mniejszej ilości kodu można zapisac to samo):

  Fragment arykułu [1] dla czasopisma “*IEEE Computer*”

  “ (…) *Torsten created an example set of classes for Person, Professor, Student, Course, and LectureRoom. He then wrote two sets of code to create, access, and manipulate objects in each of the classes. One set of code used the ODMG Java Binding and the other used the JDBC call-level interface. A summary of the results are shown in the graph below. For this exercise, 496 lines of code were needed using the ODMG Java Binding compared to 1,923 lines of code using JDBC.* “

Zalety takiego podejścia:



	LectureRoom	Course	Person	Student	Professor
ODMG	85	176	95	69	71
JDBC	360	671	195	346	351

http://www.service-architecture.com/object-relational-mapping/articles/transparent_persistence_vs_jdbc_call-level_interface.html

Wady:

- Ogólnie rzecz biorąc wydajność!
- Dosyć “skomplikowane” zapytania nie zawsze są tłumaczone najbardziej optymalnie.
- Gorsza wydajność „procedur zwrotnych” w porównaniu z procedurami składniowymi bądź „wyzwalaczami”.
- Projektowanie bazy całkowicie, w takiej technice często nie spełnia wymogow optymalności, tworzy nadmiarowe elementy.
- Zapytania, które ciężko, albo nie sposób wyrazić (albo jest to nieopłacalne ze wzg. wydajnościowych).

Kiedy, stosować?

- Aplikacje, w których wydajność nie jest najistotniejsza.
- Kiedy udostępnianie danych w postaci obiektowej jest dodatkowym atutem.
- Gdy jest zachowana należyta ostrożność i zdrowy rozsądek (np. błąd „N+1”).
- Świadomość tego, że AR nie jest lekiem na wszelkie zło.
- Gdy nie chcemy się zbytnio przepracowywać ;-)

Krótko o RoR

- **Ruby on rails** jest frameworkiem webowym, napisanym przy uzyciu architektury MVC, w obiektowym i dynamicznym języku Ruby.
- Składa się z czterech zasadniczych części:
  - **ActiveRecord** – mechanizm ORM (Object-Relational mapping) dla Ruby, odpowiada za tworzenie modeli w architekturze MVC
  - ActionPack – biblioteka zawierająca klasy ActionController i ActionView, które odpowiadają za tworzenie odpowiednio kontrolerów i widoków
  - ActiveSupport – zbiór użytecznych dodatków do standardowej biblioteki Ruby, zawiera m.in. rozszerzenia klas String czy Time
  - ActionMailer – biblioteka służąca do wysyłania wiadomości email

Przykład użycia

- Jest moduł ActiveRecords, która stanowi trzon wzorca AR. Klasą bazową jest ActiveRecord::Base
- Istnieje szereg konwencji nazewniczych. Zgodnie z zasadą “conventions over configuration”, w duchu, której powstał framework Ruby on Rails.
- Nie ma konieczności uzywania ActiveRecord wyłącznie z Ruby on Rails! (można go dodać jako osobmy moduł do praktycznie każdej aplikacji języka Ruby)

Kwestie kompatybilności

- Na dzień dzisiejszy framework moduł ActiveRecord w RoR zapewnia obsługę następujących relacyjnych DMBS'ów:
  - DB2
  - Firebird
  - FrontBase
  - MySQL
  - OpenBase
  - Oracle
  - Postgresql
  - SQL Lite
  - SQL Server
  - Sybase

Konwencje nazewnicze

Zgodnie z hasłem *“convention over configuration”* ActiveRecord oferuje nam od samego początku ujednolicony sposób zapisu:

- Nazwa tabeli, w bazie składa się z rzeczownika w liczbie mnogiej. (Odmiana 'notacji wegierskiej': wyrazy oddzielone są znakiem “_”, wszystko małą litera)
  np. employee → employees, event → events .. vertex → vertices, person -> people
  (Wszystko dzięki ActiveSupport::Inflector, który jest bardzo rozbudowany)
- W przypadku relacji “wiele do wielu” wazny jest porzadek leksykograficzny nazw!
  (np. “students_notes” jest **ŹLE**, ale “notes_students” **OK**)
- Klucz glowny tabeli jest domyslnie reprezentowany jako id (typ Integer) (naturalnie mozna z niego zrezygnować, albo zmienic typ)

Konwencje nazewnicze c.d

- Klucz obcy ma sklada sie z nazwy tabeli do ktorej klucz jest kierowany oraz sufiksu “_id” (np. “employee_id”, “student_id”)
- Nazwy klas wstepuja w liczbie pojedynczej oraz są pisane w notacji CamelCase (Np. Class Employee, Class Student)
- Nazwy plików z to nazwy klas pisane malymi literami, w notacji z “_” (Np. Class Employee → employee.rb, oraz class UserPhoto → user_photo.rb)
- .. oraz kilka innych, w chwili obecnej mniej istotnych

UWAGA: nie ma potrzeby korzystania z tych konwencji!
Mają one na celu zminimalizowanie początkowych konfiguracji, a nie ich wyeliminowanie. W klasie ActiveRecord::Base istnieje bardzo wiele metod pozwalających “wyłączenie” wszelkich domyślnych ustawień, albo zdefiniowanie własnych.

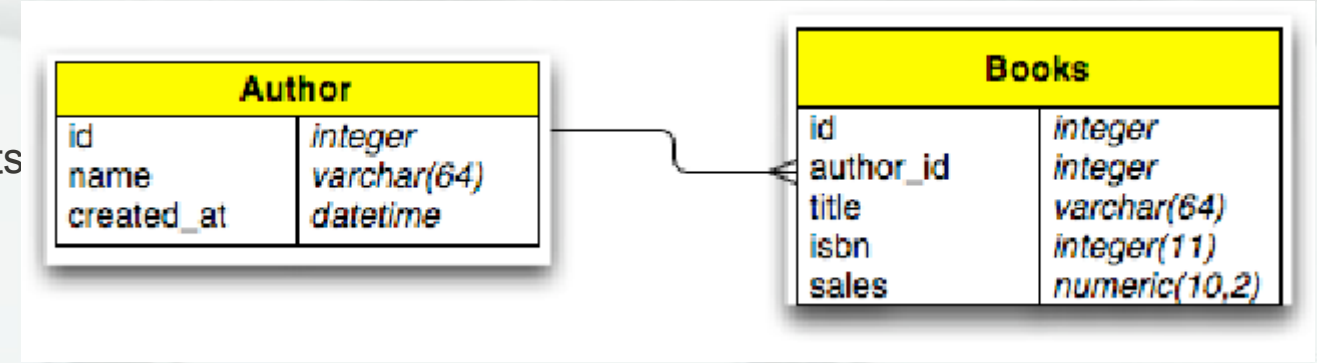
Definiowanie modelu

- Każda tabele można stworzyć:
  - „ręcznie” przy użyciu poleceń w SQL
  - Przy użyciu migracji (Ruby jest bardzo rozbudowany, w kwestii „meta-programownia” co umożliwia łatwy opis zależności między obiektami)
    - Stan bazy może być rozbudowywany
    - W każdej chwili można się cofnąć do poprzedniego stanu
    - Można ustalać „zależności” między migracjami

Przykład migracji:

Przykład 3: *„Przykład migracji”*

```
class CreateAuthors < ActiveRecord::Migration
  def self.up
    create_table :authors do |t|
      t.column :name, :string
      t.column :created_at, :datetime
    end
    create_table :books do |t|
      t.column :author_id, :integer # t.references :authors
      t.column :title, :string, :limit => 64
      t.column :isbn, :decimal, :percision => 11
      t.column :sales, :decimal, :percision => 10, :scale => 2, :default => 0
    end
  end
  def self.down
     drop_table :accounts
     drop_table :books
  end
end
```

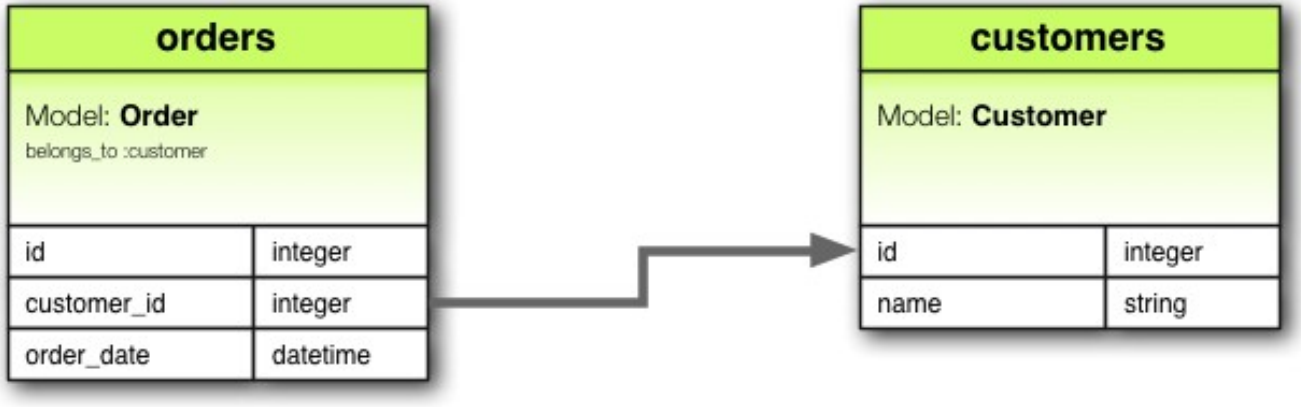


Połączenia między modelami

ActiveRecord, w Ruby on Rails umożliwia zasadniczo tworzenie połączeń między tabelami na siedem różnych sposobów:

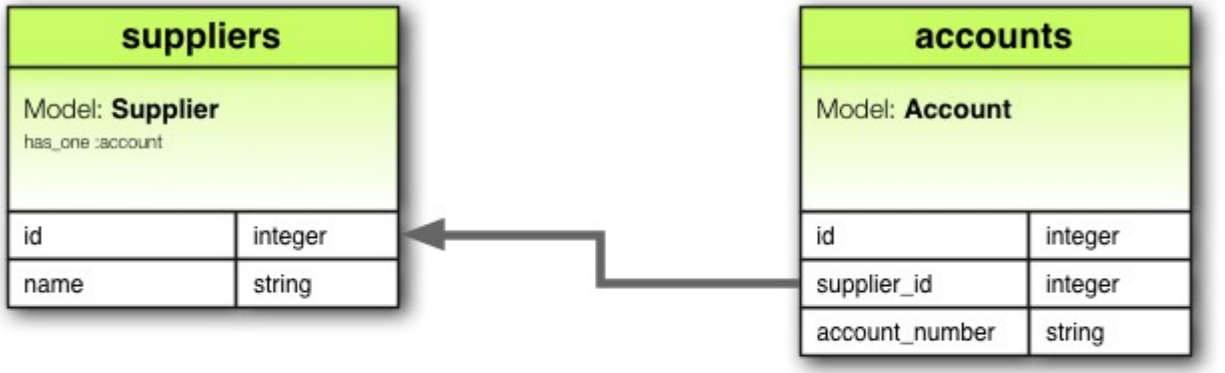
- belongs_to (relacja 1-1 odniesienie do tabeli)
- has_one (tak samo jak powyżej, ale, w drugą stronę)
- has_many (relacja 1-n)
- has_many :through (podobnie jak wyżej, ale poprzez inny model)
- has_one :through (tak jak 1-n, ale z pośrednią tabelą)
- has_and_belongs_to_many (relacja typu n-m)
- :polimorphic => true (relacje polimorficzne, różnego rodzaju, istniejące wirtualnie)

1. Relacja “belongs_to”



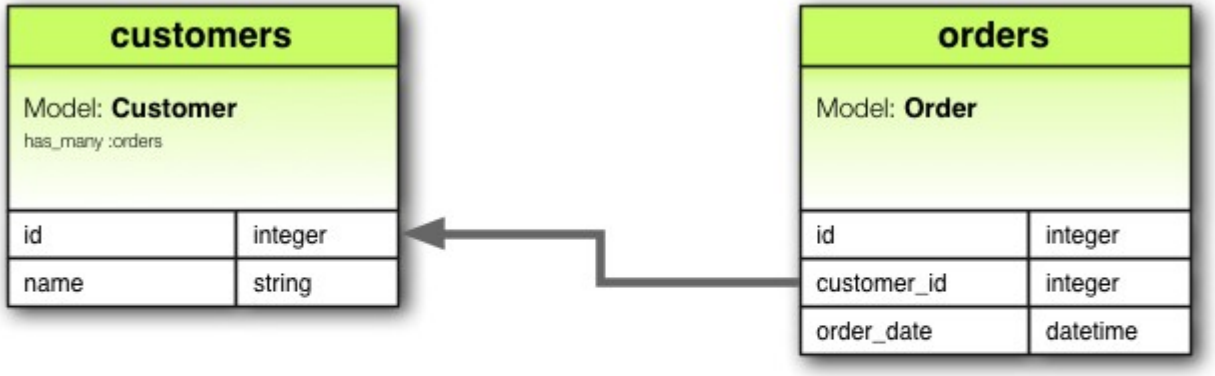

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

2. Relacja typu “has_one”



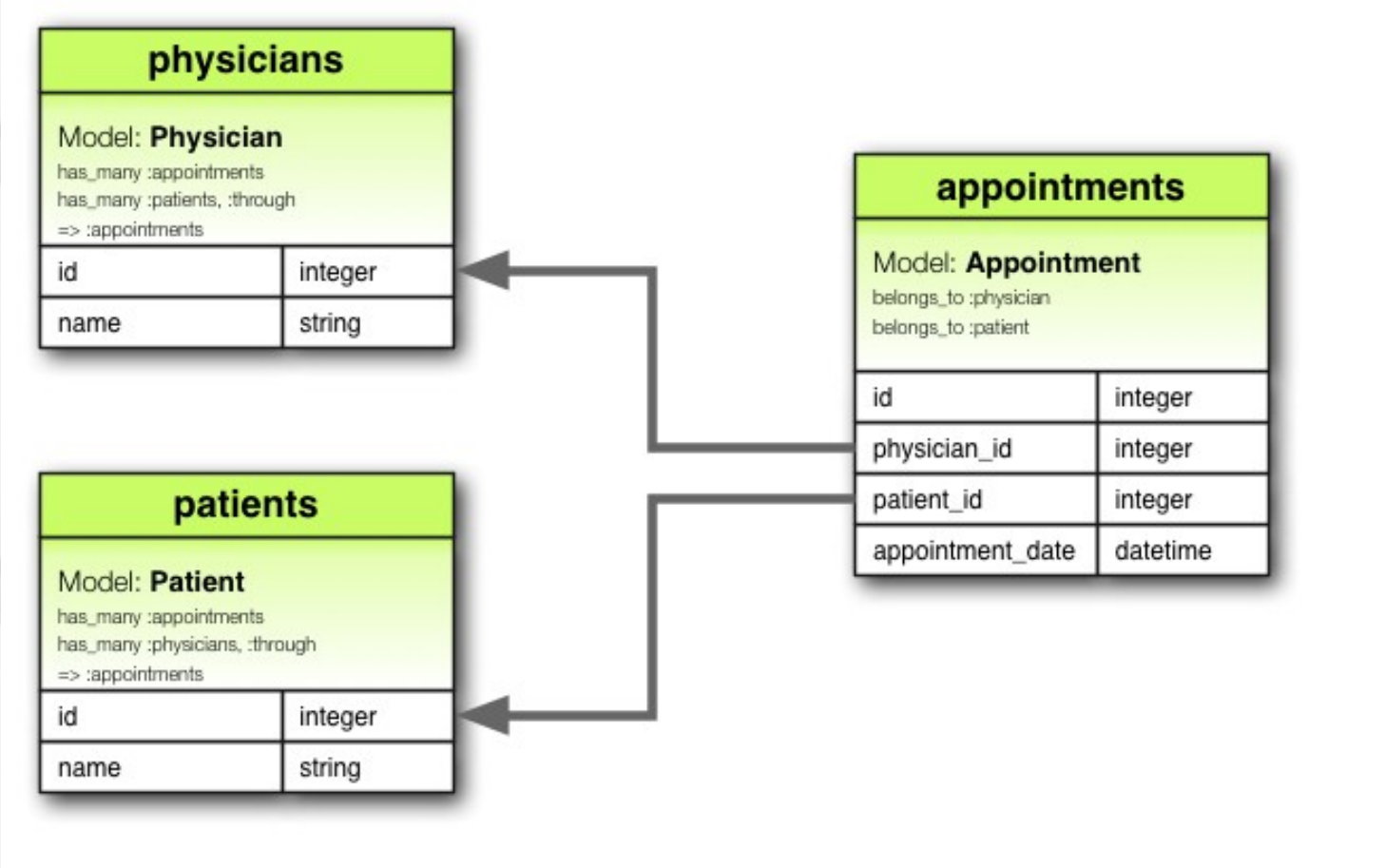

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

3. Relacja “has_many”




```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

4. Relacja “has_many :through”



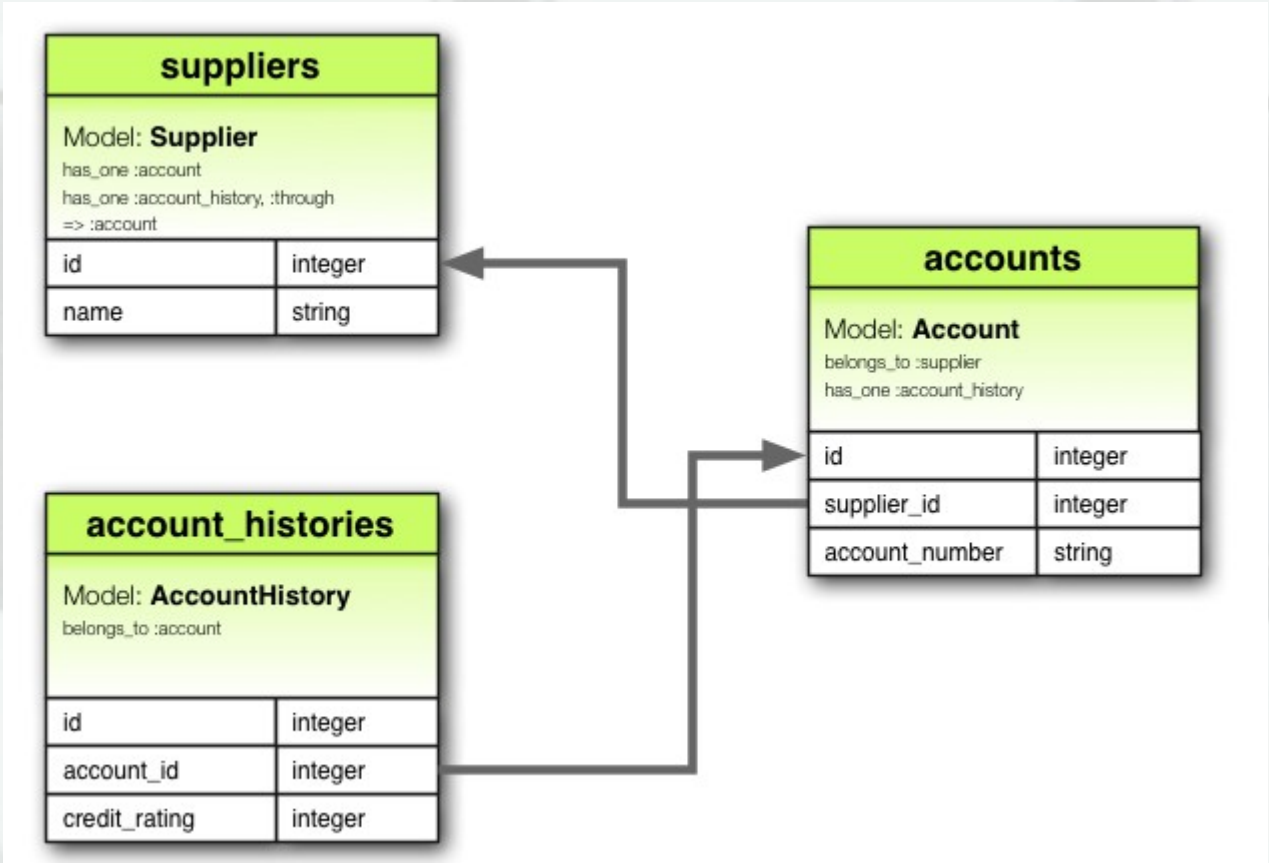
4. Relacja “has_many :through” (kod)

```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```

5. Relacja “has_one :through”



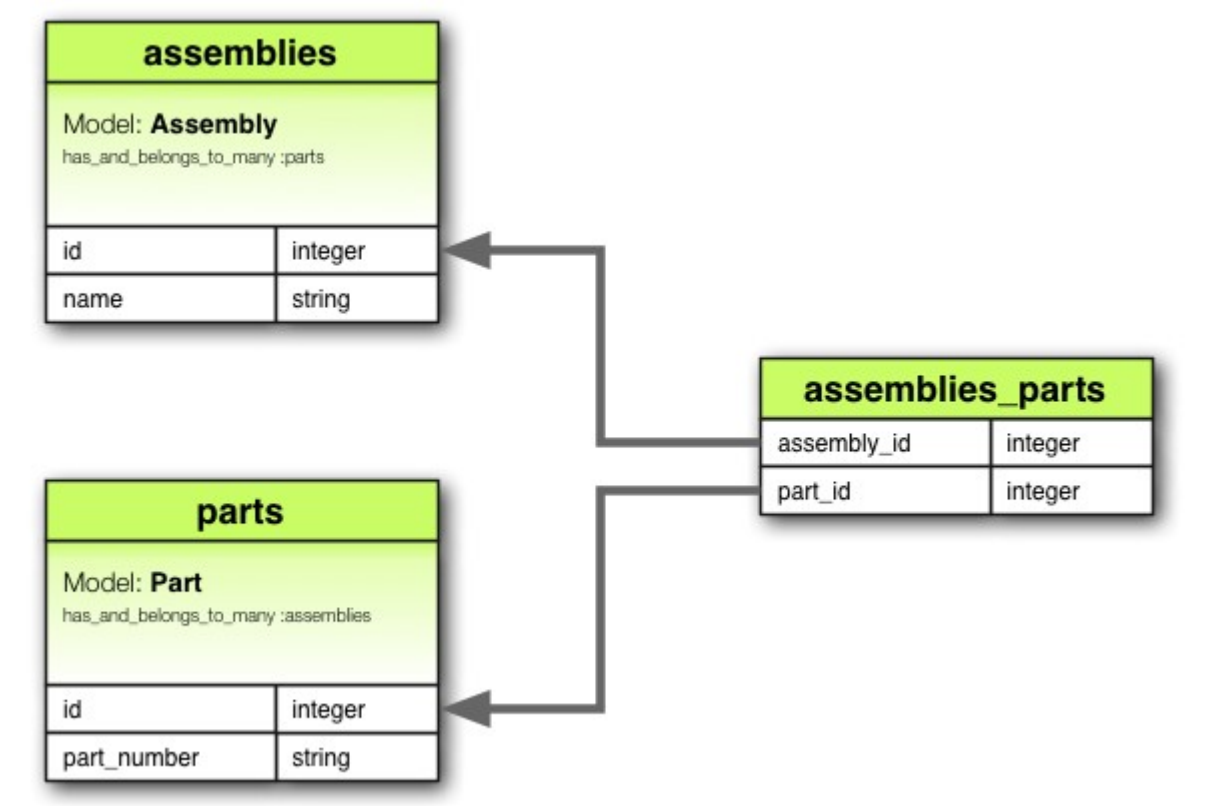
5. Relacja “has_one :through” (kod)

```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

6. Relacja “habtm”

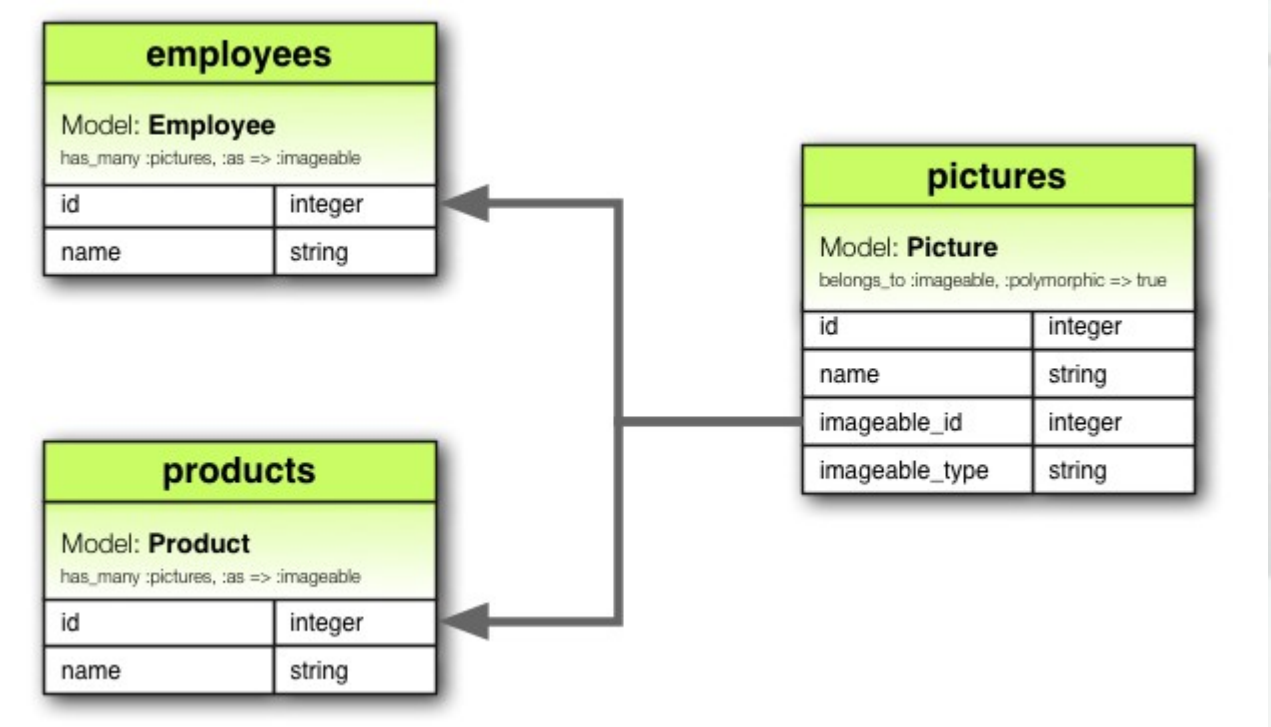


6. Relacja “habtm” (kod)

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

Przykład relacji polimorficznej



Przykład relacji polimorficznej c.d

```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

Tworzenie nowego rekordu

- **Przykład 4:** *„Tworzenie nowego obiektu klasy User”*

```
User.create(:first_name => 'Jamie') # Tworzenie nowego obiektu

# Tworzenie tablicy obiektów
User.create([{ :first_name => 'Jamie' }, { :first_name => 'Jeremy' }])

# Przekazanie bloku
User.create(:first_name => 'Jamie') do |u|
  u.is_admin = false
end

# Tworzenie kilku obiektów i wywołanie bloku dla każdego po kolei
User.create([{ :first_name => 'Jamie' }, { :first_name => 'Jeremy' }]) do |u|
  u.is_admin = false
end
```

Wyszukiwanie

- Do wyszukiwania obiektu w danym modelu używa się głównie metody: find(*args)

  Przykład 5: *„Wyszukiwanie na różne sposoby”*

  *# Zwraca pierwszy rekord z: „SELECT * FROM people”*
- Person.find(:first)
- Person.find(:first, :conditions => ["user_name = ?", user_name])
- Person.find(:first, :order => "created_on DESC", :offset => 5)
  *# Zwróci ostatni rekord z „SELECT * FROM people”*
- Person.find(:last, :conditions => ...)
- Person.find(:all, :conditions => ["category IN (?)", categories], :limit => 50)

Wyszukiwanie c.d

Przykład 5 c.d:

- Person.find(:all, :conditions => { :friends => ["Bob", "Steve"] }
- Person.find(:all, :offset => 10, :limit => 10)
- Person.find(:all, :group => "category")
- Author.find(:all, :select => [:id, :name], :include => [:books])

```
# SQL:
SELECT authors.`id` AS t0_r0, authors.`name` AS t0_r1, books.`id`
AS t1_r0, books.`title` AS t1_r1, books.`isbn` AS t1_r2, books.`sales`
AS t1_r3
FROM authors
LEFT OUTER JOIN authors_books ON
authors_books.author_id = authors.id
LEFT OUTER JOIN books ON books.id = authors_books.book_id
```

Wyszukiwanie, parametry

- Parametr (*args), może przyjąć jedną z poniższych:

- :conditions – np. "administrator = 1" albo ["user_name = ?", username]
- :order - np."created_at DESC, name".
- :group – Nazwa atrybutu do pogrupowania przez GROUP BY
- :limit – To samo co limit,w SQL
- :offset – Jak wyżej. Np. Wartość 5, przeskoczy wiersze 0 -> 4
- :joins – Złączenia. Składnia albo SQL np "LEFT JOIN comments ON comments.post_id = id" albo tablica asocjacyjna z nazwą tabeli. Dodatkowo można ustawić parametr :readonly
- :select – Domyślnie jest ustawione „*”, określa atrybuty
- :readonly – Blokada zwróconych rekordów przez zapisem
- :lock – Fragment SQL jak np. "FOR UPDATE" albo "LOCK IN SHARE MODE”

Wyszukiwanie przez SQL

- Niekiedy potrzebna jest możliwość wyrażenia czegoś poprzez „goły” SQL, wówczas na ratunek przychodzi metoda find_by_sql:

Przykład 4:

- Post.find_by_sql "SELECT p.title, c.author FROM posts p, comments c WHERE p.id = c.post_id"
  #zwróci: [#<Post:0x36bff9c @attributes={"title"=>"Ruby Meetup", "first_name"=>"Quentin"}>, ...]
- Post.find_by_sql ["SELECT title FROM posts WHERE author = ? AND created > ?", author_id, start_date] #better way
  #zwróci: [#<Post:0x36bff9c @attributes={"first_name"=>"The Cheap Man Buys Twice"}>, ...]

Kasowanie rekordów

- **Przykład 6**: *„Kasowanie poszczególnych/wszyzstkich wierszy”*

```
Todo.delete(1) # Kasowanie pojedyńczego rekordu

todos_ids = [1,2,3]
Todo.delete(todos_ids) # kasowanie kolekcji

Post.delete_all("person_id = 5 AND (category = 'Something' OR
category = 'Else')"
```

Aktualizacja rekordów

Aktualizacji można dokonywać na kilka sposobów:

Przykład 7: *„Aktualizacja”*

- Person.update(15, { :user_name => 'Samuel', :group => 'expert' }) *# Aktualizacja pojedyńczego rekordu*
- Billing.update_all("author = 'David'", "title LIKE '%Rails%'", :order => 'created_at', :limit => 5)
- foo = Foo.all(:first)
  foo.update_attribute(:bar, „baz value”)

  foo[:bar] = „baz value” # foo.bar = „baz value”
  foo.save!

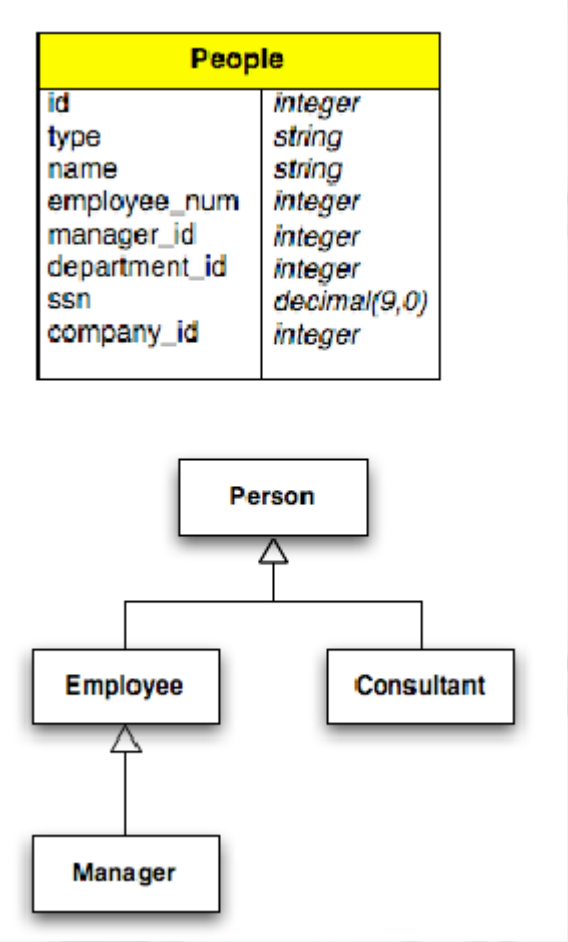
Single-table-inheritance

```
class Person < ActiveRecord::Base
end

class Employee < Person
  belongs_to :manager
  belongs_to :department
end

class Manager < Employee
  has_many :employees
end

class Consultant < Person
  belongs_to :company
end
```



Single-table-inheritance (kod)

Przykład 8: *„Kod do poprzedniego slide'u”*

```
bob = Employee.create(:name => “Bob”)
mary = Manager.create(:name => “Mary”)
tim = Consultant.create(:name => “Tim”)
bob.manager = mary
bob.save
mary.employees.map(&:name) # [‘Bob’]
jane = Manager.create(:name => ‘Jane’)
Manager.find(:all).map(&:name) # [‘Mary’, ‘Jane’]
bob.manager = jane
jane.employees.map &:name # [‘Bob’]
Person.find(:all).map &:name # [‘Bob’, ‘Mary’, ‘Tim’, ‘Jane’]
```

Pozostałe operacje na modelu

Istnieje oczywiście szereg innych operacji:

Przykład 9: *„Różne metody”*

Metody magiczne:

- User.find_by_first_name_and_salary(„wojtek”, 15 000)

Metody zależne od stanu:

- User.find_or_create_by_name('Bob', :age => 40) { |u| u.admin = true }

Sprawdzanie stanu obiektu:

- Person.column_names # => ['id', 'name', 'surname', 'age' ....]
- Person.has_key?(:salary)
- Person.exists?(:name => "Paweł")

Tworzenie transakcji

Active record umożliwa też tworzenie transakcji:

Przykład 10: „Transakcje zagnieżdzone”

```
User.transaction do
    User.create(:username => 'foo')
    User.transaction do
      User.create(:username => 'bar')
      raise ActiveRecord::Rollback
    end
end
```

Agregacja i zliczanie

- Category.find(:all, :select => 'categories.name, COUNT(posts.id) AS posts_count', :joins => :posts :group => 'categories.id, categories.name HAVING COUNT(posts.id) >= 10')

- Person.count(:conditions => "age > 26 AND job.salary > 60000", :include => :job)
- Person.count(:all, :conditions => "age > 26")
- Person.maximum('age')
- Person.sum('salary')
- Product.count_by_sql "SELECT COUNT(*) FROM sales s, customers c WHERE s.customer_id = c.id"

Funkcje zwrotne, czyli “callbacks”

- Z każdym modelem można związać funkcje zwrotne reagujące na dane zdarzenia.
- Do takich funkcji należą:
  - before_save
  - before_create
  - after_create
  - after_save
  - after_commit
  - before_validation
  - after_validation

Funkcje zwrotne c.d

Przykład 11: *„Działanie funkcji zwrotnych”*

```
class Subscription < ActiveRecord::Base
  before_create :record_signup

  private
  def record_signup
    self.signed_up_on = Date.today
  end
end

class Firm < ActiveRecord::Base
  before_destroy { |r| Client.destroy_all "client_of = #{r.id}" }
end
```

- Funkcje zwrotne są dziedziczone przez klasy

Ustalanie poprawności modelu

W celu zapewnienia poprawności można do każdego atrybutu przypisać tzw. validator, który również jest funkcją zwrotną.

Są to między innymi funkcje:

- validates_acceptance_of
- validates_associated
- validates_format_of
- validates_length_of
- validates_numericality_of
- validates_presence_of
- validates_size_of
- validates_uniqueness_of

Ustalanie poprawności c.d

Przykład 12: *„Przykład działania walidacji”*

```
class Person < ActiveRecord::Base
    validates_acceptance_of :eula, :message => "must be abided"
    validates_uniqueness_of :login_name
    validates_presence_of :first_name
    validates_length_of :smurf_leader, :is => 4, :message => "papa is
spelled with %d characters... don't play me."
end

p = Person.new(:smurf_leader => „Johny”, :login_name => „foobar”)
p.valid? # => false
p.error.messages # => [:smurf_leader => „papa is spelled with 5
characters... don't play me”, :login_name => „Attribute login_name is not
unique!”]
```

- Funkcje sprawdzające poprawność można oczywiście stworzyć samemu.

Mixin'y acts_as_...

- W języku Ruby nie ma 'wielokrotnego dziedziczenia', ale istnieją tzw. domieszki (ang. mixins). Do dowolnej klasy możemy włączać metody. Przykładami są tzw. acts_as_... (np. acts_as_nested_set, acts_as_state_machine, acts_as_list, acts_as_graph itd.)

  Przykład 13: *„Działanie acts_as_tree”*:

```
create_table :groups do |t|
   t.column :name, :string
   t.column :parent_id, :integer
end

class Group < ActiveRecord::Base
   acts_as_tree :order => :name
end

root = Group.create(:name => ‘root’)
root.children.create(:name => ‘Child 1’)
root.children.create(:name => ‘Child 2’)
```

Koniec

Dziękuję za uwagę :)

Bibliografia:

- http://ar.rubyonrails.org/
- *„Advanced Rails Recipes: 84 New Ways to Build Stunning Rails Apps”, Mike Clark, Pragmatic bookshelf, 2008*
- *http://en.wikipedia.org*